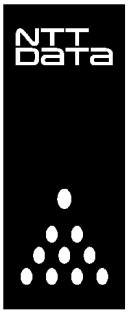


# TERASOLUNA Batch Framework for Java

## 機能説明書

第 3.1.0 版

株式会社 NTT データ

本ドキュメントを使用するにあたり、以下の規約に同意していただく必要があります。同意いただけない場合は、本ドキュメント及びその複製物の全てを直ちに消去又は破棄してください。

1. 本ドキュメントの著作権及びその他一切の権利は、NTT データあるいは NTT データに権利を許諾する第三者に帰属します。

2. 本ドキュメントの一部または全部を、自らが使用する目的において、複製、翻訳、翻案することができます。ただし本ページの規約全文、および NTT データの著作権表示を削除することはできません。

3. 本ドキュメントの一部または全部を、自らが使用する目的において改変したり、本ドキュメントを用いた二次的著作物を作成することができます。ただし、「TERASOLUNA Batch Framework for Java (機能説明書)」あるいは同等の表現を、作成したドキュメント及びその複製物に記載するものとします。

4. 前2項によって作成したドキュメント及びその複製物を、無償の場合に限り、第三者へ提供することができます。

5. NTT データの書面による承諾を得ることなく、本規約に定められる条件を超えて、本ドキュメント及びその複製物を使用したり、本規約上の権利の全部又は一部を第三者に譲渡したりすることはできません。

6. NTT データは、本ドキュメントの内容の正確性、使用目的への適合性の保証、使用結果についての的確性や信頼性の保証、及び瑕疵担保義務も含め、直接、間接に被ったいかなる損害に対しても一切の責任を負いません。

7. NTT データは、本ドキュメントが第三者の著作権、その他如何なる権利も侵害しないことを保証しません。また、著作権、その他の権利侵害を直接又は間接の原因としてなされる如何なる請求(第三者との間の紛争を理由になされる請求を含む。)に関しても、NTT データは一切の責任を負いません。

本ドキュメントで使用されている各社の会社名及びサービス名、商品名に関する登録商標および商標は、以下の通りです。

Terasoluna は、株式会社 NTT データの登録商標です。

その他の会社名、製品名は、各社の登録商標または商標です。

本書は、TERASOLUNA Batch Framework for Java ver 3.1.0 に対応しています。

# 目次

# TERASOLUNA Batch Framework for Java ver 3.x

## ■ 概要

### ◆ アーキテクチャ概要

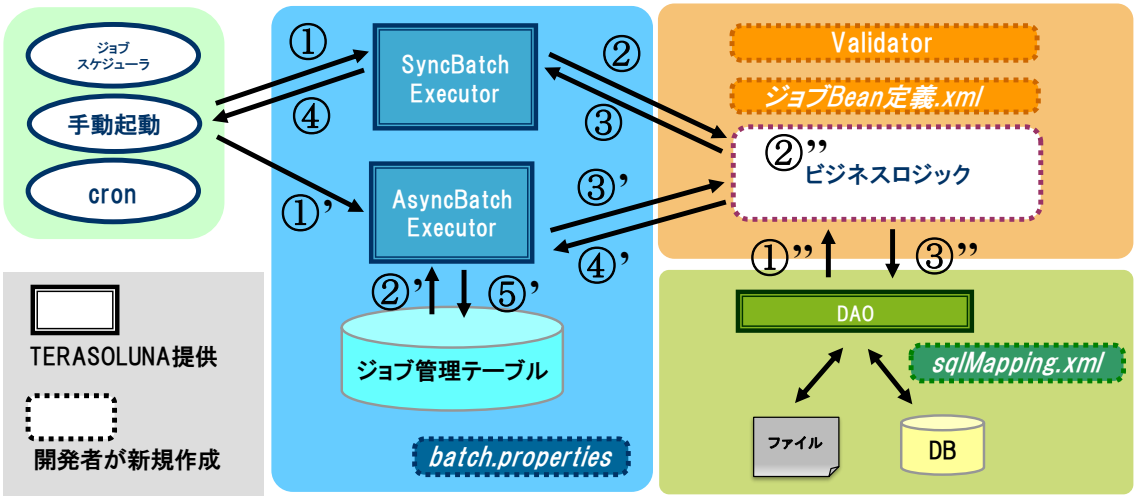
- TERASOLUNA Batch Framework for Java ver 3.x（以下、フレームワークと略す）は、バッチシステムを構築するための実行基盤、共通機能を提供するフレームワークである。
- TERASOLUNA Server Framework for Java（Web版、Rich版）の開発者が最小限の学習コストでバッチ開発を習得することが可能である。
- 本フレームワークはTERASOLUNA Batch Framework for Java、Spring Framework、iBATISのベースフレームワークとしている。

### ◆ 機能概要

- BL-01 同期型ジョブ実行機能 **→BL-01参照**
  - SyncBatchExeccutorを利用し、ジョブスケジューラ、起動用のシェルからジョブを実行することができる。
- BL-02 非同期型ジョブ実行機能 **→BL-02参照**
  - AsyncBatchExecutorを利用し、ジョブ管理テーブルに登録されたジョブを非同期に実行することができる。
- BL-03 トランザクション管理機能 **→BL-03参照**
  - フレームワークがトランザクションを管理する方式を提供する。
  - ビジネスロジック内でトランザクションを管理できる方式を提供する。
- BL-04 例外ハンドリング機能 **→BL-04参照**
  - ビジネスロジック内で例外が発生した場合、発生した例外をハンドリングし、ジョブ終了コードを設定することできる。
- BL-05 ビジネスロジック実行機能 **→BL-01, BL-03, BL-04参照**
  - 開発者は、フレームワークが提供するインタフェースを実装、または抽象クラスを継承してビジネスロジックを作成する。
  - ビジネスロジックの戻り値がそのままジョブ終了コードとなる
- BL-06 データベースアクセス機能 **→BL-06参照**
  - データベースアクセスを簡易化するDAOを提供する。
- BL-07 ファイルアクセス機能 **→BL-07参照**
  - CSV 形式、固定長形式、可変長形式ファイルの入出力機能を提供する。
- BL-08 ファイル操作機能 **→BL-08参照**
  - ファイルのコピーや削除・結合などといった機能を提供する。
- BL-09 メッセージ管理機能 **→BL-09参照**
  - アプリケーションユーザなどに対して表示する文字列(メッセージリソース)を、定義できる。

- AL-036 バッチ更新最適化機能 **→AL-036参照**
  - バッチ更新を行う前にSQLの発行順を最適化する機能を提供する。
- AL-041 入力データ取得機能 **→AL-041参照**
  - DBやファイルから入力データを取得する機能を提供する
- AL-042 コントロールブレイク機能 **→AL-042参照**
  - コントロールブレイク処理を行うためのユーティリティを提供する。
- AL-043 入力チェック機能 **→AL-043参照**
  - 入力データ取得機能を使用した際に、DBやファイルから取得したデータ1件毎に入力チェックを行う機能を提供する。

## ◆ 概念図




## ◆ 解説

- 同期型ジョブ実行

① 同期ジョブを実行する場合 SyncBatchExecutor を利用しジョブを起動する。
② 起動時のパラメータより、ジョブを構成するジョブ Bean 定義ファイルを読み込み、該当するビジネスロジックを呼び出す。
③ ビジネスロジックの戻り値が返却される。
④ ビジネスロジックの戻り値がジョブ終了コードとして返却される。

- 非同期型ジョブ実行

①' 非同期ジョブを実行する場合 AsyncBatchExecutor を利用しジョブを起動する。
②' ジョブの起動パラメータをジョブ管理テーブルから取得する。
③' ジョブ実行用のスレッドを立ち上げ、ジョブを構成するジョブ Bean 定義ファイルを読み込み、該当するビジネスロジックを呼び出す。
④' ビジネスロジックの戻り値が返却される。
⑤' ビジネスロジックの戻り値がジョブ終了コードとしてジョブ管理テーブルに登録され、ジョブステータスが処理済みに更新される。

- ビジネスロジックの実行（同期、非同期共通）

①'' ビジネスロジック内で DAO を利用し、ファイル/DB からデータを抽出する。
②'' 起動時のパラメータや①''で取得したデータをもとに処理を行う。
③'' 処理結果は DAO を利用し、ファイル/DB へ出力される。

## ◆ 動作確認環境

- 対応JDK
  - Oracle Sun JDK5.0/6.0
- 対応データベース
  - Oracle 11g
  - PostgreSQL 8.x

## ◆ 参照ライブラリ

- 依存するTERASOLUNAのライブラリ

| TERASOLUNA ライブラリ名 | 説明 | バージョン |
|---|---|---|
| terasoluna-commons.jar | ユーティリティ機能など共通機能を提供する | 2.0.3.1 |
| terasoluna-dao.jar | DAO インタフェースを提供する | 2.0.3.1 |
| terasoluna-ibatis.jar | OR マッピングツール iBatis を利用した、データベースアクセス機能を提供する | 2.0.3.1 |
| terasoluna-validator.jar | 入力チェック機能を提供する | 2.0.3.1 |
| terasoluna-logger-1.0.0.jar | 汎用ログ・汎用例外メッセージログ出力機能を提供する | 1.0.0 |
| terasoluna-filedao-2.0.3.1.jar | ファイルアクセス機能を提供する | 2.0.3.1 |
| terasoluna-batch-update-1.1.0.jar | バッチ更新最適化機能を提供する | 1.1.0 |
| terasoluna-collector-1.0.0.jar | 入力データ取得機能、コントロールブレイク機能、入力チェック機能を提供する | 1.0.0 |

● 依存するオープンソースライブラリ一覧

| オープンソースライブラリ名 | バージョン |
|---|---|
| aspectjweaver.jar | 1.6.11 |
| cglib-nodep.jar | 2.1_3 |
| commons-beanutils.jar | 1.7.0 |
| commons-collections.jar | 3.2 |
| commons-dbcp.jar | 1.2.2 |
| commons-digester.jar | 1.8 |
| commons-jxpath.jar | 1.3 |
| commons-lang | 2.3 |
| commons-logging.jar | 1.1.1 |
| commons-pool.jar | 1.3 |
| commons-validator.jar | 1.3.1 |
| ibatis.jar | 2.3.4.726 |
| jakarta-oro.jar | 2.0.8 |
| log4j.jar | 1.2.15 |
| spring.jar | 2.5.6.SEC01 |
| spring-modules-validation.jar | 0.8 |

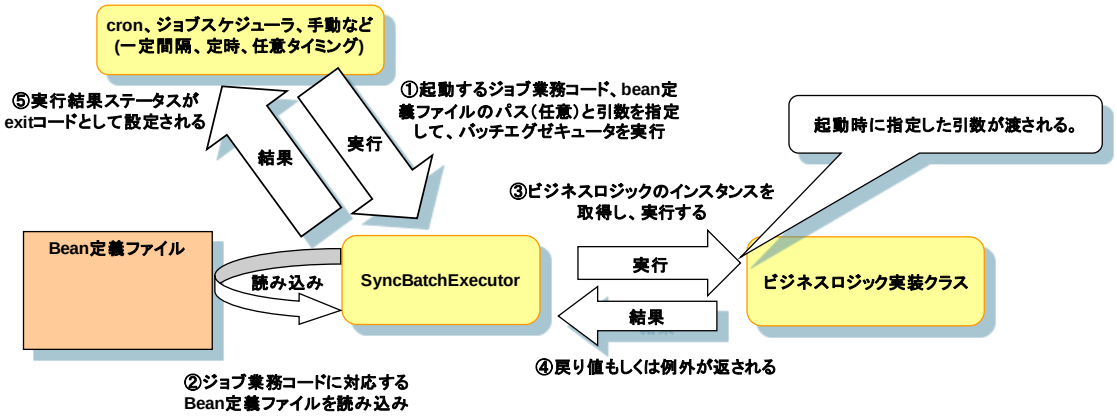
# BL-01 同期型ジョブ実行機能

## ■ 概要

### ◆ 機能概要

- 同期型ジョブ実行機能として SyncBatchExecutor クラスを提供する
- バッチ処理 ID を直接指定して特定のバッチを 1 件実行する
- 単一のスレッドで実行し、処理終了後にプロセス終了する

### ◆ 概念図




### ◆ 解説

- 同期型ジョブの起動から終了までの流れ

① 起動するジョブ業務コードと引数を指定して、バッチエグゼキュータを実行する。
  - ➢ 指定のジョブ業務コードに対応するビジネスロジックを単体実行する。
  - ➢ 実行パラメータは引数もしくは環境変数に設定する。

| 実行時引数 | 環境変数 | 名称 | 説明 |
|---|---|---|---|
| 第 1 引数 | JOB_APP_CD | ジョブ業務コード | 実行するジョブの ID（必須） |
| 第 2～21 引数 | JOB_ARG_NM1～20 | 引数 | ジョブに引き渡す引数 |

※実行時引数と環境変数を両方とも指定した場合は、実行時引数が優先される。

② ジョブ業務コードに対応する Bean 定義ファイルを読み込む。
- 第 1 引数のジョブ業務コードから、「ジョブ業務コード」＋「.xml」の名称である Bean 定義ファイルを読み込む。

例）ジョブ業務コードを B000001 と設定した場合、
B000001.xml がフレームワークに読み込まれる。

③ ビジネスロジックのインスタンスを取得し、実行する。
- 読み込んだ Bean 定義ファイル（コンテキスト）から、ジョブ業務コード＋「BLogic」の名称であるビジネスロジックのインスタンスを取得する。

例）ジョブ業務コードを B000001 した場合、B000001BLogic クラスの
インスタンスを取得し、実行する。

④ 戻り値もしくは例外が返される。
⑤ 実行結果ステータスがジョブ終了コード（exit コード）として設定される。

- プロパティファイルの設定値
  - ApplicationResource.properties ファイルに設定されたプロパティファイルが読込みされる。デフォルトは batch.properties。
  - batch.properties にフレームワークに関する設定を記述されている。
    - ✧ 業務要件によってカスタマイズする場合は、batch.properties ファイルの値を変える。

| プロパティキー | デフォルト値 | 説明 |
|---|---|---|
| beanDefinition.admin.classpath | beansDef/ | 管理用 Bean 定義ファイルを配置するクラスパス |
| beanDefinition.admin.default | AdminContext.xml | 管理用 Bean 定義（基本部）ファイル |
| beanDefinition.business.classpath | beansDef/ | 業務用 Bean 定義ファイルを配置するクラスパス<br># 業務用 bean 定義ファイルパスのはバッチ実行時に java の-D で指定して渡すことも可能。 |
| messageAccessor.default | msgAcc | メッセージソースアクセサの Bean 名 |

# ■ 使用方法

## ◆ コーディングポイント

- ジョブ起動シェルスクリプトの作成
  - SyncBatchExecutor クラスを実行するにはシェル（UNIX）またはバッチファ

イル（windows）を実装する必要がある。
以下、Bourne Shell をもとに説明する。

- クラスパスファイル（classpath.sh）の設定を行う。

```
sh_classpath=${sh_classpath}:${lib_path}/aopalliance-1.0.jar"
sh_classpath=${sh_classpath}:${lib_path}/commons-lang-2.3.jar"
  … 略 …
sh_classpath=${sh_classpath}:${lib_path}/terasoluna-filedao-2.0.3.1.jar"
sh_classpath=${sh_classpath}:${lib_path}/terasoluna-validator-2.0.3.1.jar"
```

- パラメータを実行時に渡す場合
  - ✧ 業務ジョブコード：B000001
  - ✧ 実行パラメータ：[2, 3, 4]

```
#!/bin/sh
batch_dir=/business/job/sh/batch

# 共通 CLASSPATH 定義シェル実行
class_path="${batch_dir}/classpath.sh"

# バッチ起動
java –cp ${class_path} jp.terasoluna.fw.batch.executor.SyncBatchExecutor B000001 2 3 4
```

上記で設定したクラスパスを-cp に指定する。

業務ジョブコードを第一引数に設定する。引数はスペースを空け設定する

※ 必要に応じて-Xms や-Xmx などのオプションを設定し、実行時ヒープサイズを変更する。

- パラメータを環境変数で指定した場合
  - ✧ 業務ジョブコード：B000001
  - ✧ 実行パラメータ：[2, 3, 4]

```
#!/bin/sh
batch_dir=/business/job/sh/batch

# 共通 CLASSPATH 定義シェル実行
class_path="${batch_dir}/classpath.sh"

SET JOB_APP_CD=B000001
SET JOB_ARG_NM1=2
SET JOB_ARG_NM2=3
SET JOB_ARG_NM3=4

# バッチ起動
java –cp ${class_path} jp.terasoluna.fw.batch.executor.SyncBatchExecutor
```

業務ジョブコードや引数を環境変数に設定する。

※ 必要に応じて-Xms や-Xmx などのオプションを設定し、実行時ヒープサイズ

を変更する。

- ジョブ Bean 定義ファイルの設定
  - ジョブ Bean 定義ファイル名は「ジョブ業務コード」＋「.xml」にする。
    - ✧ SyncBatchExecutor クラスに渡されたジョブ業務コードによって、同名の Bean 定義ファイルを読み込まれる。
  - アノテーションの有効
    - ✧ ビジネスロジック内でアノテーションを利用できるように context:annotation-config を設定する。
  - 共通コンテキストのインポート
    - ✧ ビジネスロジックで利用する共通の Bean 定義（ファイル系 DAO やデフォルト例外ハンドラ）を利用する場合は、インポートする。
  - データソース定義のインポート
    - ✧ ビジネスロジックで利用するデータソース関連の Bean 定義を利用する場合はそのデータソースの定義ファイルの参照を記述する。
  - コンポーネントスキャンの定義
    - ✧ コンポーネントスキャンで定義されたパッケージからビジネスロジッククラスが自動的にロードされる。

例）B000001.xml 実装例

```
・・・
<!-- アノテーションによる設定 -->
<context:annotation-config/>

<!-- 共通コンテキスト-->
<import　resource=" classpath:beansDef/commonContext.xml" />

<!-- データソース設定　-->
<import　resource="classpath:beansDef/dataSource.xml" />

<!-- コンポーネントスキャン設定 -->
<context:component-scan base-package="jp.terasoluna.batch.sample.b000001" />
・・・
```

コンポーネントスキャンのベースパッケージに業務のパッケージを指定すると設定する。

- ビジネスロジックの実装
  - BLogic インタフェースを実装する
    - ✧ トランザクションをフレームワーク側で管理する場合は抽象クラスである AbstractTransactionBLogic クラスを継承すること。（詳細は後述するトランザクション管理機能を参照すること）
  - クラス名の宣言に@Componet アノテーションを付与し、先のコンポーネントスキャンの対象にする。
  - Autowired アノテーションを利用して Bean 定義ファイルに定義した Bean を

フィールドにインジェクションできる。

- execute メソッドを実装する。
  - ✧ 引数として渡される BLogicParam は、起動時に引数もしくは環境変数として設定された値が渡される。

例）B000001BLogic 実装例（JDK5 または JDK6）

```
@Component
public class B000001BLogic implements BLogic {

  @Autowired
  protected QueryDAO queryDAO = null;

  public int execute(BLogicParam param) {

    //業務処理
    //終了コードの返却
    return 0;
  }
}
```

@Component アノテーションを付与することにより、自動的に DI コンテナの管理対象となる。

ジョブ Bean 定義ファイル内に定義された Bean をフィールドに設定したい場合 @Autowired アノテーションを利用する。
型が同じ Bean が自動で設定されるが、複数 Bean が同じ型で定義されており、ByName でインジェクションしたい場合は@Qualifier と併用して利用すること

例）Bean 定義に「queryDAO_1」と「queryDAO_2」が定義されており、queryDAO_1 をフィールドにインジェクションしたい場合
@Autowired
@ Qualifier(“queryDAO_1”)
private QueryDAO queryDAO = null
とする。

BLogic の戻り値がジョブ終了コードとして返却される。

例）B000001BLogic 実装例（JDK6 のみ）

```
@Component
public class B000001BLogic implements BLogic {

  @Resource(“queryDAO”)
  private QueryDAO queryDAO = null;

  public int execute(BLogicParam param) {

    //業務処理
    //終了コードの返却
    return 0;
  }
}
```

JDK6.0 の場合、@Resource アノテーションが利用すれば ByName でインジェクション可能

BLogic の戻り値がジョブ終了コードとして返却される。

### ◆ 構成クラス

| | クラス名 | 概要 |
|---|---|---|
| 1 | jp.terasoluna.fw.batch.executor.BatchExecutor | バッチエグゼキュータインタフェース。 |
| 2 | jp.terasoluna.fw.batch.executor.AbstractBatchExecutor | 同期バッチエグゼキュータ抽象クラス。 |
| 3 | jp.terasoluna.fw.batch.executor.SyncBatchExecutor | 同期バッチエグゼキュータ。<br>指定のジョブ業務を実行する。 |

## ■ 関連機能

- 『BL-06 データベースアクセス機能』
- 『BL-07 ファイルアクセス機能』
- 『BL-08 ファイル操作機能』
- 『BL-03 トランザクション管理機能』
- 『BL-04 例外ハンドリング機能』

## ■ 使用例

- 機能網羅サンプル(terasoluna-batch-functionsample)
- チュートリアル(terasoluna-batch-tutorial)

## ■ 備考

- @JobComponent アノテーションについて。
  - ビジネスロジックのクラス名の宣言に付与する@Component の代わりに用いることが出来るアノテーションである。
  - @JobComponent アノテーションを使用する事により、ビジネスロジックのクラス名を「ジョブ ID＋BLogic」にする必要がなくなる。

例）B000001BLogic 実装例（JDK5 または JDK6）

```
@ JobComponent(jobId = " B000001")
public class SampleBLogic implements BLogic {

  …以下略
```

@JobComponent はジョブ ID を付与して使用する。

  - 上記のように設定した場合、ビジネスロジック実行時にはジョブ ID に指定した「B000001.xml」が使用される。

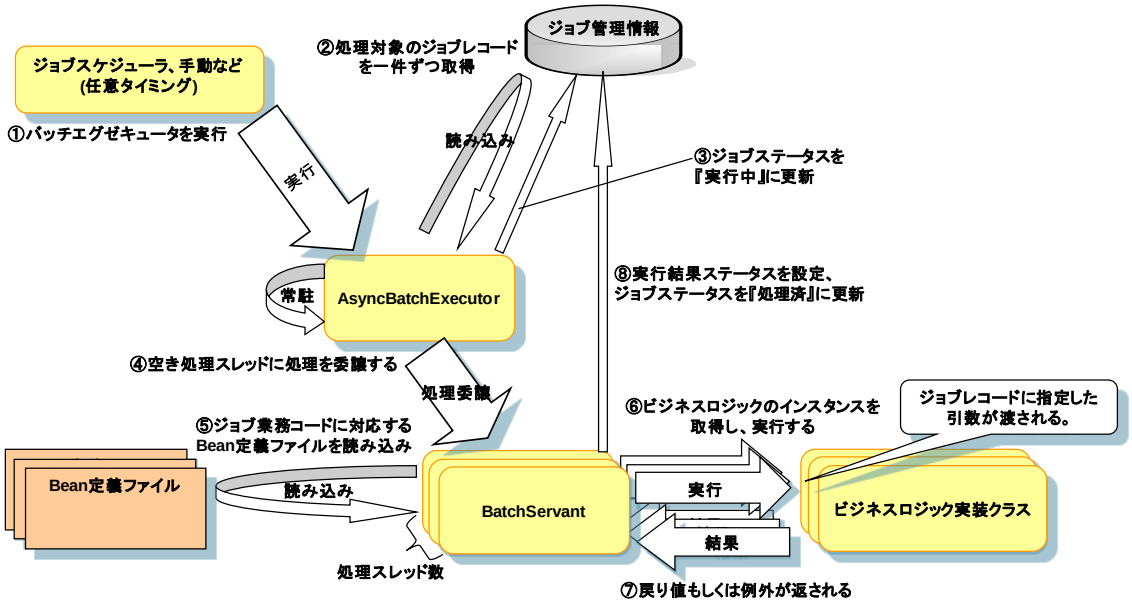
# BL-02 非同期型ジョブ実行機能

## ■ 概要

### ◆ 機能概要

- 非同期型ジョブ実行機能として AsyncBatchExecutor クラスを提供する
- AsyncBatchExecutor は常駐プロセスとして起動し、ジョブ管理テーブルに実行対象ステータスのレコードが登録される毎にジョブを実行する
- メインスレッドとは別の実行スレッドで処理が行われる

### ◆ 概念図




### ◆ 解説

- 非同期型バッチの起動から終了までの流れ
  - ① 非同期型バッチエグゼキュータを実行する。
  - ② 処理対象のジョブレコードを一件ずつ取得する。
  - ③ ジョブステータスを『実行中』に更新する。
  - ④ 空き処理スレッドに処理を委譲する。
    - ➢ BatchThreadPoolTaskExecutor を利用して、ジョブ管理テーブルに登録されたジョブを複数スレッドで順次実行する。
    - ➢ 実行スレッド数の調整はシステム用 Bean 定義ファイルに記述する。

```
<!-- バッチ実行用スレッドプールタスクエグゼキュータ -->
<bean id="batchTaskExecutor"
   class="jp.terasoluna.fw.batch.executor.concurrent.BatchThreadPoolTaskExecutor">
      <property name="corePoolSize" value="1" />
      <property name="maxPoolSize" value="2" />
</bean>
```

| プロパティ | 説明 |
|---|---|
| corePoolSize | コアスレッド数を設定する |
| maxPoolSize | スレッドの最大許容数を設定する |

⑤ ジョブ業務コードに対応する Bean 定義ファイルを読み込む。
- ジョブ管理テーブルに登録された「job_app_cd」カラムからジョブ業務コードを取得し、「ジョブ業務コード」＋「.xml」の名称である Bean 定義ファイルを読み込む。

⑥ ビジネスロジックのインスタンスを取得し、実行する。
⑦ 戻り値もしくは例外が返される。
⑧ 実行結果ステータスを設定、ジョブステータスを『処理済』に更新する。

- ジョブ管理テーブル内容

デフォルトのジョブ管理テーブルの構成は以下の通りである。

| 項番 | 属性名 | カラム名 | 必須 | 概要 |
|---|---|---|---|---|
| 1 | ジョブシーケンスコード | job_seq_id | ○ | ジョブの登録順にシーケンスから払い出す。 |
| 2 | ジョブ業務コード | job_app_cd | ○ | 実行するビジネスロジックに対応するID |
| 3 | 引数 1 | job_arg_nm1 | | ビジネスロジックに渡す引数 |
| | … | … | | |
| 22 | 引数 20 | job_arg_nm20 | | ビジネスロジックに渡す引数 |
| 23 | ビジネスロジック戻り値 | blogic_app_status | | ビジネスロジックの戻り値 |
| 24 | ジョブステータス | cur_app_status | ○ | ジョブの状態を表すステータス<br>ジョブのステータスは以下の 3 つとなる。<br>未実施：0<br>実行中：1<br>処理済み：2 |
| 25 | 登録時刻 | add_data_time | | ジョブ登録時刻 |
| 26 | 更新時刻 | upd_data_time | | ジョブ更新時刻 |

※ジョブ管理テーブルのカラム名は変更することができる。変更する場合はフレームワーク内部で発行される SQL 文も併せて変更すること。

- プロパティファイルの設定
  - ApplicationResource.properties ファイルに設定されたプロパティファイルが読込みされる。デフォルトは batch.properties。
  - batch.properties にフレームワークに関する設定を記述されている。
    - ✧ 業務要件によってカスタマイズする場合は、batch.properties ファイルの値を変える。

| プロパティキー | デフォルト値 | 説明 |
|---|---|---|
| beanDefinition.admin.classpath | beansDef/ | 管理用 Bean 定義ファイルを配置するクラスパス. |
| beanDefinition.admin.default | AdminContext.xml | 管理用 Bean 定義（基本部）ファイル |
| beanDefinition.business.classpath | beansDef/ | 業務用 Bean 定義ファイルを配置するクラスパス.<br># 業務用 bean 定義ファイルパスのはバッチ実行時に java の-D で指定して渡すことも可能。 |
| messageAccessor.default | msgAcc | メッセージソースアクセサの Bean 名 |
| systemDataSource.queryDAO | adminQueryDAO | システム用 DAO 定義<br>※ジョブ管理情報テーブルを参照する |
| systemDataSource.updateDAO | adminUpdateDAO | システム用 DAO 定義<br>※ジョブ管理情報テーブルを k 更新する |
| systemDataSource.<br>transactionManager | adminTransactionManager | システム用トランザクションマネージャ定義 |
| polling.interval | 3000 | ジョブ管理テーブルにジョブがない、もしくは実行スレッド空きがない状態でのポーリング実行間隔（ミリ秒） |
| executor.jobTerminateWaitInterval | 3000 | Executor のジョブ終了待ちチェック間隔（ミリ秒） |
| executor.endMonitoringFile | /tmp/batch_terminate_file | Executor の常駐モード時の終了フラグ監視ファイル（フルパスで記述） |
| batchTaskExecutor.default | batchTaskExecutor | Executor のスレッドタスクエグゼキュータの Bean 名 |
| batchTaskExecutor.batchServant | batchServant | スレッド実行用の BatchServant クラスの Bean 名 |

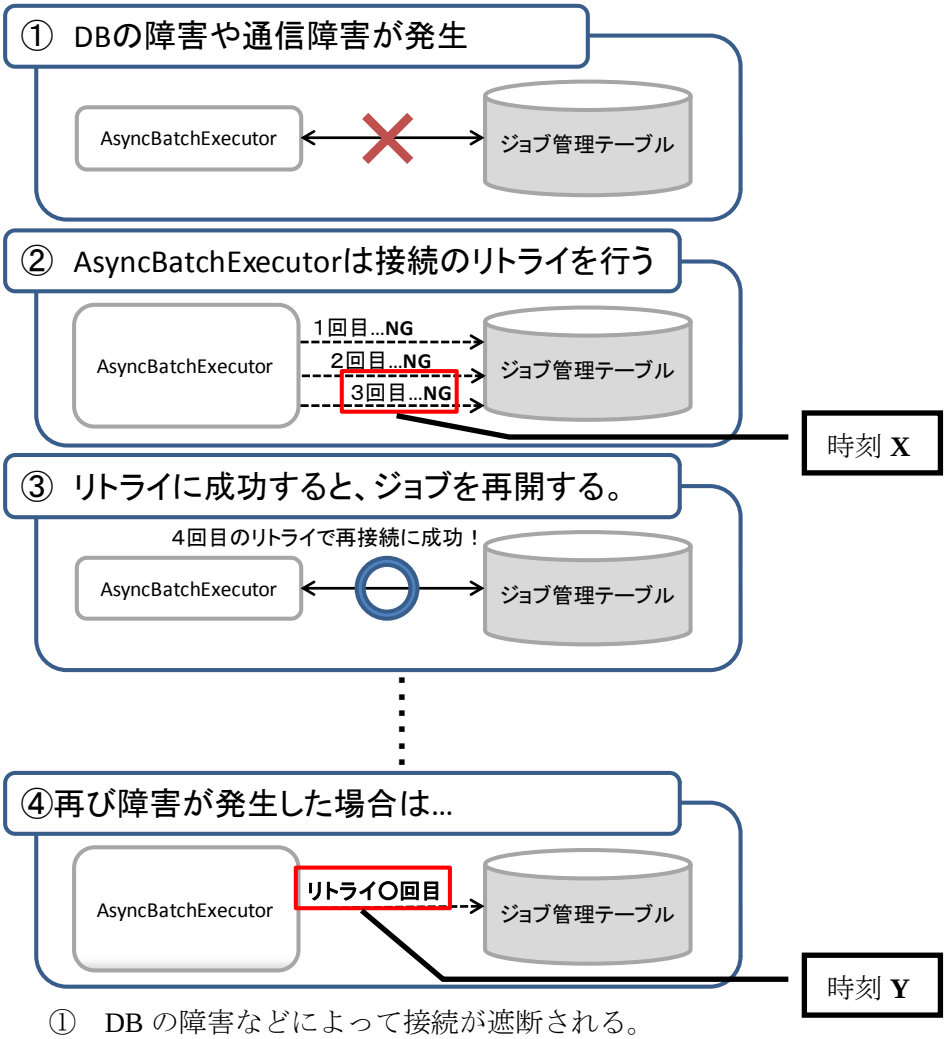
| | | |
|---|---|---|
| batchTaskExecutor.dbAbnormalRetryMax | 0 | データベース異常時のリトライ回数 |
| batchTaskExecutor.dbAbnormalRetryInterval | 20000 | データベース異常時のリトライ間隔（ミリ秒） |
| batchTaskExecutor.dbAbnormalRetryReset | 600000 | データベース異常時のリトライ回数をリセットする前回からの発生間隔（ミリ秒） |

- 非同期型バッチエグゼキュータの強制終了
  - 終了ファイルによる強制終了
    - ✧ 非同期型バッチエグゼキュータは周期的にプロパティ（executor.endMonitoringFile）に設定されているファイルをチェックしている。非同期型バッチエグゼキュータを終了させたい場合は、プロパティ値と同名のファイルを配置すること。
    - 例）executor.endMonitoringFile=/tmp/batch_terminate_file と設定されている場合、windows 環境であれば C:¥tmp フォルダに batch_terminate_file というファイル名を配置すれば、非同期型バッチエグゼキュータは終了する。（※ファイルの内容はなくてもよい）
    - ✧ 終了ファイルチェック後、ジョブステータスを確認し、実行中のジョブが終了次第、非同期型バッチエグゼキュータを終了する。

- 非同期型バッチエグゼキュータの異常終了
  - 終了ファイル以外の異常終了
    - ✧ コマンドラインからの Ctrl+C 命令や、ハードウェア故障によるプロセスダウン処理で非同期型バッチエグゼキュータは異常終了する。
    - ✧ 実行中のジョブも途中で終了し、そのジョブの処理はロールバックされる。
  - DB サーバがシャットダウンした場合の異常終了
    - ✧ 非同期型バッチエグゼキュータは周期的にジョブ管理テーブルをチェックしているが、DB サーバが途中でシャットダウンした場合は通信ができなくなる。その場合、デフォルトの設定では非同期型バッチエグゼキュータもプロセスを終了する。
    - ✧ 実行中のジョブは途中で終了し、そのジョブの処理はロールバックされる

- 非同期型バッチエグゼキュータのリトライ機能
  - DB サーバのシャットダウンなどにより、通信が切断された際も、プロパティの値を変更することで、DB サーバへ接続をリトライする事ができる。
  - 次ページで動作イメージを掲載して解説をする。

➢ リトライ機能のイメージ図は以下のようになる。
batchTaskExecutor.dbAbnormalRetryMax = 10
batchTaskExecutor.dbAbnormalRetryInterval=20000(デフォルト値)
batchTaskExecutor.dbAbnormalRetryReset=600000(デフォルト値)
と設定したとする。




① DB の障害などによって接続が遮断される。
② AsyncBatchExecutor は batchTaskExecutor.dbAbnormalRetryInterval に設定された間隔(今回は 20000 ミリ秒)で接続のリトライを試みる。
**##この時３回目のリトライを行った時刻を「X」とする##**
③ ４回目のリトライで接続に成功し、ジョブを再開する。
④ 再び障害が発生し、DB との接続が遮断され、AsyncBatchExecutor は再びリトライを試みる。 **##この時の時刻を「Y」とする##**
時刻 Y から時刻 X を差し引いた値が batchTaskExecutor.dbAbnormal RetryReset に設定された値(今回は 600000 ミリ秒)を上回っていた場合は、リトライ回数をリセットし、１回目のリトライとしてカウントする。
逆に 600000 ミリ秒を下回っていた場合は、前回に続く４回目のリトライとしてカウントする。

# ■ 使用方法

## ◆ コーディングポイント

- ジョブ起動シェルスクリプトの作成
  - AsyncBatchExecutor クラスを実行するにはシェル（UNIX）またはバッチファイル（windows）を実装する必要がある。
    以下、Bourne Shell をもとに説明する。

  - クラスパスファイル（classpath.sh）の設定を行う。

```
sh_classpath=${sh_classpath}:${lib_path}/aopalliance-1.0.jar"
sh_classpath=${sh_classpath}:${lib_path}/commons-lang-2.3.jar"
  … 略 …
sh_classpath=${sh_classpath}:${lib_path}/terasoluna-filedao-2.0.3.1.jar"
sh_classpath=${sh_classpath}:${lib_path}/terasoluna-validator-2.0.3.1.jar"
```

  - 非同期用バッチエグゼキュータの起動
    - ✧ クラスパス（classpath.sh）をインポートして非同期バッチエグゼキュータを行う。

```
#!/bin/sh
batch_dir=/business/job/sh/batch

# 共通 CLASSPATH 定義シェル実行
class_path="${batch_dir}/classpath.sh"

# バッチ起動
java –cp ${class_path} jp.terasoluna.fw.batch.executor.AsyncBatchExecutor
```

※ 必要に応じて-Xms や-Xmx などのオプションを設定し、実行時ヒープサイズを変更する。

- プロパティファイルの設定
  - 同期型ジョブ実行機能と同様

- Bean 定義ファイルの設定。
  - 同期型ジョブ実行機能と同様

- ビジネスロジックの実装
  - 同期型ジョブ実行機能と同様

### ◆ 構成クラス

| | クラス名 | 概要 |
|---|---|---|
| 1 | jp.terasoluna.fw.batch.executor.AbstractJobBatchExecutor | 非同期バッチエグゼキュータ抽象クラス。 |
| 2 | jp.terasoluna.fw.batch.executor.AsyncBatchExecutor | 非同期バッチエグゼキュータ。<br>常駐プロセスとして起動し、ジョブ管理テーブルに登録されたジョブを取得し、ジョブの実行を BatchServant クラスに移譲する。 |
| 3 | jp.terasoluna.fw.batch.executor.concurrent.BatchServant | バッチサーバントインタフェース。非同期バッチエグゼキュータから呼ばれ、指定されたジョブシーケンスコードからジョブを実行する。 |
| 4 | jp.terasoluna.fw.batch.executor.concurrent.BatchServantImpl | バッチサーバント実装クラス。<br>非同期バッチエグゼキュータから呼ばれ、指定されたジョブシーケンスコードからジョブを実行する。 |
| 5 | jp.terasoluna.fw.batch.util.JobUtil | ジョブ管理情報関連ユーティリティ。<br>主にフレームワークの AbstractJobBatchExecutor から利用されるユーティリティ。 |

### ◆ 拡張ポイント

- ジョブ管理テーブルのカスタマイズ
  以下のような業務要件によってジョブ管理テーブルをカスタマイズすることが可能である。
  - グループ ID によるジョブのノード分割処理
  - 優先度カラムによるジョブ実行順序の制御

## ■ 関連機能

なし

## ■ 使用例

- 機能網羅サンプル(terasoluna-batch-functionsample)
- チュートリアル(terasoluna-batch-tutorial)

## ■ 備考

なし

# BL-03 トランザクション管理機能

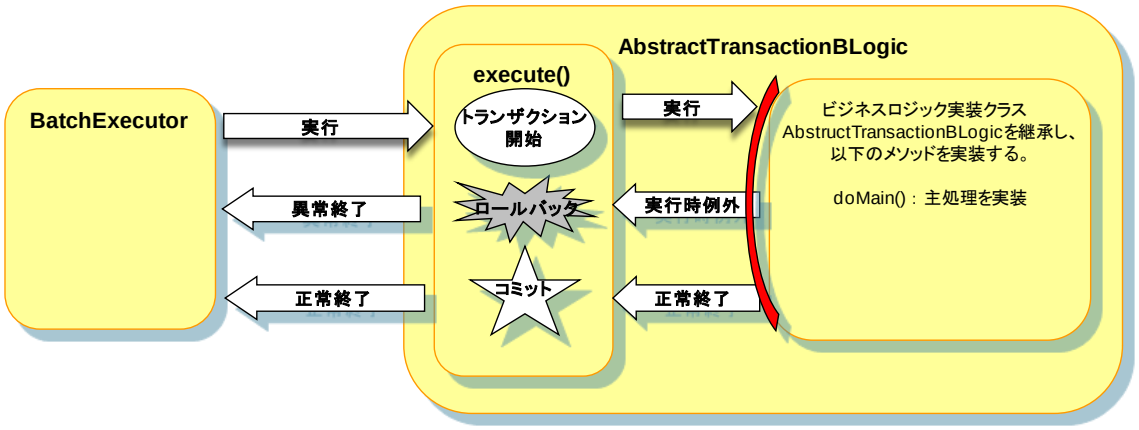
## ■ 概要

### ◆ 機能概要

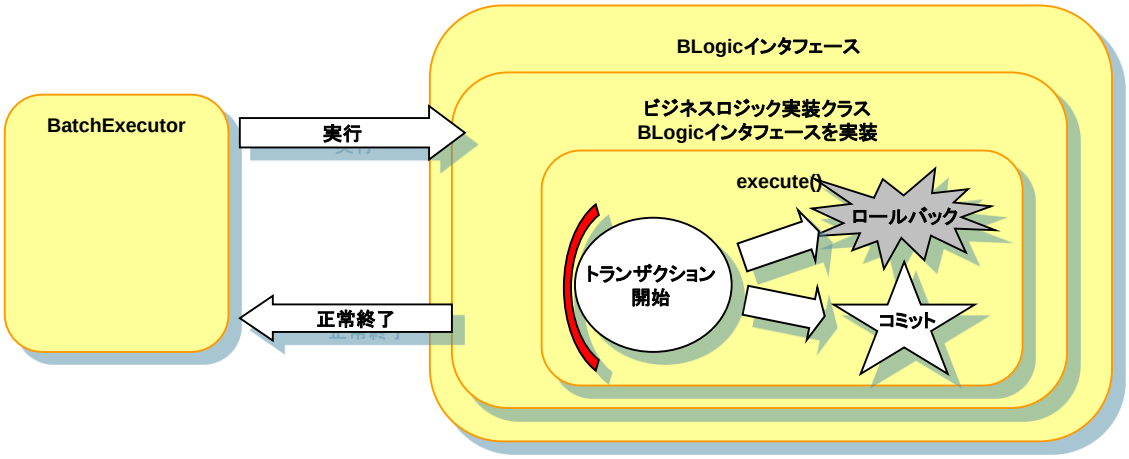
- フレームワークで以下の二つのトランザクションモデルを提供する。
  開発者は、業務要件に応じてトランザクションモデルを選択する。
  - フレームワークがトランザクションを管理するモデル
    - ✧ 1 ビジネスロジック 1 トランザクションで完結するモデル.。通常はこちらのモデルを選択する。AbstractTransactionBLogic を継承する
  - ビジネスロジックで任意にトランザクションを管理するモデル
    - ✧ 複雑なトランザクション管理を必要とする場合に選択する。BLogic インタフェースを実装する

### ◆ 概念図

- フレームワークがトランザクションを管理するモデルの場合



- ビジネスロジックで任意にトランザクションを管理するモデルの場合




### ◆ 解説

- バッチ実行タイプ（同期型・非同期型）にも関わらず、トランザクション管理は上記の 2 種類である。
  - フレームワークがトランザクションを管理するモデルの場合
    - AbstractTransactionBLogic を継承してビジネスロジックを実装する
    - フレームワークがトランザクション制御を行うため、開発者はコードを実装する必要がない。
    - ビジネスロジック開始時にトランザクションが開始され、終了時にコミットされる。実行時例外発生時、ロールバックされる。
  - ビジネスロジックで任意にトランザクションを管理するモデルの場合
    - BLogic インタフェースを実装し、任意でトランザクションを管理する
    - フレームワークはトランザクション管理しないため、開発者が業務要件により、トランザクションの開始・終了またコミットやロールバックを行う。

## ■ 使用方法

### ◆ コーディングポイント

- Bean 定義ファイルの設定
  - 以下はトランザクション管理機能の両方のパターンと対して共通の設定である。
  - bean 定義ファイルの設定
    - バッチ実行タイプ（同期型・非同期型）に関わらず以下の Bean 定義ファイルの設定を行う。
    - DataSource の設定を行う。詳細はデータアクセス機能を参照すること。
    - トランザクションマネージャは、Spring が提供する

DataSourceTransactionManager を使用する。

✧ DataSourceTransactionManager は、単一のデータソースに対してトランザクションを実行するトランザクションマネージャである。
※複数のデータソースの扱いに関しては後述する備考を参照すること。

```
<!-- DataSource の設定。 -->
<bean id="dataSource" class="……">……</bean>

<!-- 単一のデータソース向けのトランザクションマネージャ。 -->
<bean id="transactionManager"
  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

- ビジネスロジックの実装
  - フレームワークがトランザクションを管理するモデルの場合
    - ✧ AbstractTransactionBLogic を継承する
    - ✧ AbstractTransactionBLogic クラスの doMain()をオーバライドして、業務処理の実装を行う。
    - ✧ ビジネスロジックでトランサクションをロールバックしたい場合は、実行例外である BatchException をスローする。

```
@Component
public class B000001BLogic extends AbstractTransactionBLogic {

    @Override
    public int doMain(BLogicParam param) {
      try {
        //業務処理
        … 略 …
      } catch(Exception ex) {
        throw new BatchException(ex);
      }
      return 0
    }
}
```

ビジネスロジック開始時にトランザクションが開始される。

BLogicException がスローされ、ロールバックされる。

正常終了後、コミットされ、トランザクションが終了する。

- ビジネスロジックで任意にトランザクションを管理するモデルの場合
  - ✧ BLogic インタフェースの execute()をオーバライドして業務処理の実装を行う
  - ✧ PlatformTransactionManager のフィールドを定義する
  - ✧ フレームワーク提供のユーティリティを利用し、トランサクション管理を行う。

```
@Component
public class B000001BLogic implements BLogic {

    @ Autowired
    private PlatformTransactionManager transactionManager = null;

    @Override
    public int execute (BLogicParam param) {
        TransactionStatus stat = null;
        try {
            //トランザクションを開始する
            stat = BatchUtil.startTransaction(transactionManager);

            // 業務処理
            … 略 …

            if (エラー条件) {
                BatchUtil. rollbackTransaction (transactionManager, stat);
                return 255;
            } else {
                //コミットを行う
                BatchUtil. commitTransaction (transactionManager, stat);
                return 0;
            }
        } finally {
            // トランザクションを終了させる。
            // 未コミット時はロールバックする。
            BatchUtil.endTransaction(transactionManager, stat);
        }
    }
}
```

BLogic インタフェースを実装した場合は、明示的にトランザクションを開始する。フレームワークが提供するユーティリティを利用する。

ロールバックされ、ジョブ終了コード 255 が返される。実行例外をスローし、例外ハンドラでジョブ終了コードの設定を行ってもよい。

コミットされ、正常終了し、
ジョブ終了コード 0 が返される。

コミット、ロールバックに関わらず
必ずトランザクションは終了すること。

✧ ビジネスロジック途中で 100 件毎にコミットするようなトランザクション開始・終了の繰り返しがある時に以下のように実装する。

```
@Component
public class B000002BLogic implements BLogic {

    @Autowired
    private PlatformTransactionManager transactionManager = null;

    @Override
    public int execute(BLogicParam param) {
        TransactionStatus stat = null;
        try {
          stat = BatchUtil.startTransaction(transactionManager);
          for ( int i = 0; i < =1000; i++){
            // 業務処理
            if( i % 100 == 0){
              BatchUtil. commitTransaction (transactionManager, stat);
              stat = BatchUtil.startTransaction(TransactionManager);
            }
          }
          return 0;
        } finally {
            // トランザクションを終了させる
            // 未コミット時はロールバックする
            BatchUtil.endTransaction(transactionManager, stat);
       }
    }
}
```

必ず TransactionStatus を設定すること。
設定せずにトランザクションを開始してもエラーが発生せずに処理は実行されるが、正しくコミットされない

# ■ リファレンス

## ◆ 構成クラス

| | クラス名 | 概要 |
|---|---|---|
| 1 | jp.terasoluna.fw.batch.blogic.BLogic | ビジネスロジックインタフェース。<br>任意にトランザクションを管理したい場合の BLogic インタフェースを実装すること。 |
| 2 | jp.terasoluna.fw.batch.blogic.AbstractBLogic | ビジネスロジック抽象クラス。<br>任意にトランザクションを管理したい場合の AbstractBLogic を継承すること。 |
| 3 | jp.terasoluna.fw.batch.blogic.AbstractTransactionBLogic | トランザクション管理を行うビジネスロジック抽象クラス。フレームワーク側でトランザクション管理を行いたい場合、この抽象クラスを継承し、AbstractTransactionBLogic#doMain メソッドを実装してビジネスロジックが作成する。 |
| 4 | jp.terasoluna.fw.batch.util.BatchUtil | バッチ実装用ユーティリティ。<br>各種バッチ実装にて使用するユーティリティメソッドを定義する。 |

# ■ 関連機能

- 『BL-06 データベースアクセス機能』

# ■ 使用例

- 機能網羅サンプル(terasoluna-batch-functionsample)
- チュートリアル(terasoluna-batch-tutorial)

## ■ 備考

- 複数データソースの利用について
  複数のデータソースを扱う場合、データソースの Bean 定義を複数用意する。
  - dataSource_1.xml の設定例

```
<!-- DBCP のデータソース 1 を設定する  -->
<bean id="dataSource_1" destroy-method="close"
          class="org.apache.commons.dbcp.BasicDataSource">
        …　略　…
</bean>

<bean  id="transactionManager_1"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
<property  name="dataSource"  ref="dataSource_1"  />
  <!--  複数 DB 接続を行うためにトランザクション同期を行わない設定  -->
  <property  name="transactionSynchronization"  value="2"/>
</bean>

…以下、sqlMapClient、DAO の Bean 定義を設定する…
```

  - dataSource_2.xml の設定例

```
<!-- DBCP のデータソース 2 を設定する  -->
<bean id="dataSource_2" destroy-method="close"
          class="org.apache.commons.dbcp.BasicDataSource">
        …　略　…
</bean>

<bean  id="transactionManager_2"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
<property  name="dataSource"  ref="dataSource_2"  />
  <!--  複数 DB 接続を行うためにトランザクション同期を行わない設定  -->
  <property  name="transactionSynchronization"  value="2"/>
</bean>

…以下、sqlMapClient、transactionManager、DAO の Bean 定義を設定する…
```

➢ ビジネスロジックの設定例

```
@Autowired
@Qualifier(“queryDAO_1”)
private QueryDAO queryDAO_1 = null;

@Autowired
@Qualifier(“queryDAO_2”)
private QueryDAO queryDAO_2 = null;

@Autowired
@Qualifier(“updateDAO_1”)
private UpdateDAO updateDAO_1 = null;

@Autowired
@Qualifier(“updateDAO_2”)
private UpdateDAO updateDAO_2 = null;

@Override
public int doMain(BLogicParam param) {
        … 略 …
}
```

ただし上記設定ではトランザクションは各データソースで完結するため、複数データソース全体の原子性は保証されていない。

また、複数データソースの transactionSynchronization の設定を行うと、LOB 列へのアクセス時でエラーが発生するため、LOB 列へアクセスする際は複数データソースを利用しないようにすること。

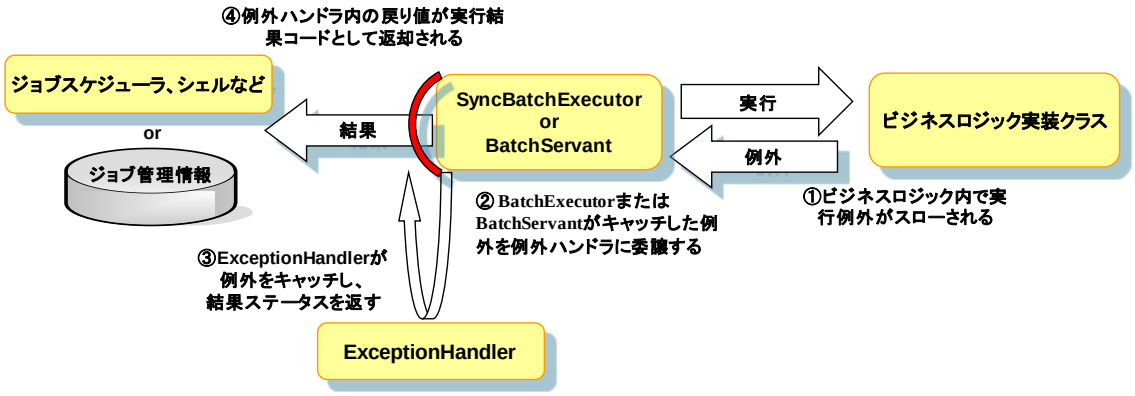
# BL-04 例外ハンドリング機能

## ■ 概要

### ◆ 機能概要

- ビジネスロジックにてスローされた実行例外をハンドリングできる機能を提供する。
- 例外ハンドリングクラスで設定された戻り値が、ジョブ終了コードとして返却される

### ◆ 概念図




### ◆ 解説

① ビジネスロジック内で実行例外がスローされる
② BatchExecutorまたはBatchServantがキャッチした例外を例外ハンドラに委譲する。
③ ExceptionHandlerが例外をキャッチし、結果ステータスを返す
- Bean定義に設定した独自の例外ハンドラが使用される。例外ハンドラが実装されていない場合はデフォルト例外ハンドラであるDefaultExceptionHandlerが使用される

④ 例外ハンドラ内の戻り値が実行結果コードとして返却される

## ■ 使用方法

### ◆ コーディングポイント

- 例外ハンドリングクラスの実装
  - 業務処理ごとに例外をハンドリングしたい場合、フレームワークが提供する

ExceptionHandler インタフェースを実装する。

- 例外ハンドリグクラス名は「ジョブ業務コード」＋「ExceptionHandler」を命名すること。
- 特定例外発生時のログ出力やジョブ終了コードを設定可能

例）B000001 のジョブの例外ハンドラクラスを作成する場合

```
@Component
public class B000001ExceptionHandler implements ExceptionHandler {
    private static Log logger = LogFactory.getLog(B000001ExceptionHandler.class);
    @Override
    public int handleThrowableStatus(Throwable e) {
        // WARN ログを出力する
        if (logger.isWarnEnabled()) {
            logger.wart(MessageUtil.getMessage("errors.exception"));
            logger.warn("An exception occurred.", e);
        }
        // ジョブ終了コードとして返却したい値を設定する
        return 100;
    }
}
```

クラス名は「ジョブ業務コード」＋「ExceptionHandler」と設定する。

# ■ リファレンス

## ◆ 構成クラス

| | クラス名 | 概要 |
|---|---|---|
| 1 | jp.terasoluna.fw.batch.exception.handler.ExceptionHandler | 例外ハンドラインタフェース。<br>独自に例外ハンドラクラスを作成する場合は ExceptionHandler インタフェースを実装する。 |
| 2 | jp.terasoluna.fw.batch.exception.handler.DefaultExceptionHandler | 例外ハンドラのデフォルト実装。<br>フレームワークがデフォルトで用意している例外ハンドラクラス。 |
| 3 | jp.terasoluna.fw.batch.exception.BatchException | バッチ例外クラス。バッチ実行時に発生した例外情報を保持する。 |

## ◆ 拡張ポイント

なし

# ■ 関連機能

なし

# ■ 使用例

- 機能網羅サンプル(terasoluna-batch-functionsample)

# ■ 備考

なし

# BL-06 データベースアクセス機能

## ■ 概要

### ◆ 機能概要

- TERASOLUNA Batch Framework for Java ver 3.x で使用するデータベースアクセス機能は、TERASOLUNA Server Framework for Java ver 2.x で使用していたデータベースアクセス機能と同一のものを利用して、データベースアクセスを行う。
- 本項目では、TERASOLUNA Batch Framework for Java ver 3.x でデータベースアクセス機能を使用する場合の TERASOLUNA Server Framework for Java ver 2.x との違いのみを説明するものとし、データベースアクセス機能の詳細な説明は別資料の「CB-01 データベースアクセス機能」の機能説明書を参照すること。

## ◆ コーディングポイント

- 本説明書でのコーディングポイントは、別資料の「CB-01　データベースアクセス機能」のコーディングポイントと異なる以下の項目についてのみ説明を行う。
  - SqlMapClientFactoryBean の Bean 定義
  - DAO の Bean 定義
  - QueryDAOiBatisImpl を使用した一覧データ取得例
  - UpdateDAOiBatisImpl を使用したデータ登録例
  - QueryRowHandleDAOiBatisImpl を使用したデータ取得例データソースの Bean 定義

  その他の項目については、「CB-01　データベースアクセス機能」を参照すること。

- データソースの Bean 定義
  - データソースの設定は Bean 定義ファイル beansDef/dataSource.xml に定義する。
  - Bean 定義例(beansDef/dataSource.xml)

```
<!-- DBCP のデータソースを設定する。　-->
<context:property-placeholder location="SqlMapConfig/jdbc.properties" />
<bean id="dataSource" destroy-method="close"
      class="org.apache.commons.dbcp.BasicDataSource">
      <property name="driverClassName" value="${driver}" />
      <property name="url" value="${url}" />
      <property name="username" value="${username}" />
      <property name="password" value="${password}" />
      <property name="maxActive" value="10" />
      <property name="maxIdle" value="1" />
      <property name="maxWait" value="5000" />
</bean>
```

設定値はプロパティファイルに切り離し、プレースホルダを利用して設定する。

  - Bean 定義で利用されるプロパティファイル例(SqlMapConfig/jdbc.properties)

```
#
# ジョブ管理テーブル DB 接続先
#
driver=org.postgresql.Driver
url=jdbc:postgresql://127.0.0.1:5432/postgres
username=postgres
password=postgres
```

- SqlMapClientFactoryBean の Bean 定義
  Spring で iBatis を使用する場合、SqlMapClientFactoryBean を使用して iBatis 設定ファイルの Bean 定義を DAO に設定する必要がある。
  SqlMapClientFactoryBean は、iBatis のデータアクセス時に利用されるメインのクラス「SqlMapClient」を管理する役割をもつ。
  iBatis 設定ファイルの Bean 定義はアプリケーション内で一つとする。
  - iBatis 設定ファイル
    "configLocation"プロパティに、iBatis 設定ファイルのコンテキストルートからのパスを指定する。
  - 単一データベースの場合は"dataSource"プロパティに、使用するデータソースの Bean 定義を設定する。
  - Bean 定義例(beansDef / dataSource.xml)

```
<!-- システム共通 SqlMapConfig 定義 -->
<bean id="sqlMapClient" class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
        <property name="configLocation" value="SqlMapConfig/SqlMapConfig.xml" />
        <property name="dataSource" ref="dataSource" />
</bean>
```

  - 複数のデータベースの場合は"dataSource"プロパティは指定せずに、"configLocation"プロパティのみ設定する。
  - Bean 定義例(beansDef / dataSource.xml)

```
<!-- システム共通 SqlMapConfig 定義 -->
<bean id="sqlMapClient" class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
        <property name="configLocation" value="SqlMapConfig/SqlMapConfig.xml" />
</bean>
```

- DAO の Bean 定義
  DAO の実装クラスは、beansDef/dataSource.xml に定義する
  Bean 定義時に DAO 実装クラスの"sqlMapClient"プロパティに iBatis 設定ファイルの Bean 定義を設定する必要がある。
  - Bean 定義例(beansDef / dataSource.xml)

```
<!-- システム共通 SqlMapConfig 定義 -->
<bean id="sqlMapClient" class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
        <property name="configLocation" value="SqlMapConfig/SqlMapConfig.xml" />
        <property name="dataSource" ref="dataSource" />
</bean>

<!-- 照会系の DAO 定義 -->
<bean id="queryDAO" class="jp.terasoluna.fw.dao.ibatis.QueryDAOiBatisImpl">
        <property name="sqlMapClient" ref="sqlMapClient" />
</bean>
```

  - 複数データベースの場合は"sqlMapClient"プロパティの設定だけでなく、"dataSource"プロパティに、DAO 実装クラスで使用するデータソースを指定する必要がある。
  - Bean 定義例(beansDef/dataSource.xml)

```
<!-- 照会系の DAO 定義 -->
<bean id="queryDAO" class="jp.terasoluna.fw.dao.ibatis.QueryDAOiBatisImpl">
        <property name="sqlMapClient" ref="sqlMapClient" />
        <property name="dataSource" ref=" dataSource " />
</bean>
```

- QueryDAOiBatisImpl を使用した一覧データ取得例
  TERASOLUNA Batch Framework for Java ver 3.x では、DB からのデータの取得の為に「AL-041 入力データ取得機能」を提供しており、QueryDAOiBatisImpl を使用したデータの取得を推奨していない。
  DB からのデータの取得を行う場合は、「AL-041 入力データ取得機能」の内容を参考にし、実装すること。

- UpdateDAOiBatisImpl を使用したデータ登録例
  UpdateDAOiBatisImpl を使用して、データベースに情報を登録する場合の設定およびコーディング例を以下に記述する。

① DAO 実装クラスを以下のように Bean 定義ファイルに定義する。

➢ Bean 定義例(beansDef/dataSource.xml)

```
<!-- 更新系の DAO 定義 -->
<bean id="updateDAO" class="jp.terasoluna.fw.dao.ibatis.UpdateDAOiBatisImpl">
        <property name="sqlMapClient" ref="sqlMapClient" />
</bean>
```

② ビジネスロジックを作成する。
Bean 定義ファイルにて設定された DAO の updateDAO.execute(String sqlID, Object bindParams)メソッドを使用する。
メソッドの引数に、「発行する SQLID」と「SQL に関連付けられるオブジェクト」を設定する必要がある。

➢ ビジネスロジック実装例

```
@Autowired
protected UpdateDAO updateDAO = null;

    public int execute(BLogicParam arg0) {
        ......
        updateDAO.execute("insertUser", bean);
        ......
    }
```

設定された DAO を使用して、
データベースにデータを登録する。

- QueryRowHandleDAOiBatisImpl を使用したデータ取得例
  「QueryDAOiBatisImpl を使用した一覧データ取得例」の項目でも説明したように、DB からのデータの取得には「AL-041 入力データ取得機能」を推奨している。
  「AL-041 入力データ取得機能」を使用した場合に、要件を満たせないような場合のみ、以下を参考にして QueryRowHandleDAOiBatisImpl を使用した DB からの取得を実装すること。
  (Bean 定義ファイルの定義方法は、UpdateDAOiBatisImpl と同様なため省略する)
  - DataRowHandler の実装

```
import jp.terasoluna.fw.dao.event.DataRowHandler;

public class SampleRowHandler implements DataRowHandler {
    public void handleRow(Object param) {
        if (param instanceof HogeData) {
            HogeData hogeData = (HogeData)param;
            // 一件のデータを処理するコードを記述
        }
    }
}
```

一件毎に handleRow メソッドが呼ばれ、引数に一件分のデータが格納されたオブジェクトが渡される。

一件のデータを元に更新処理を行うのであれば、あらかじめ DataRowHandler に UpdateDAO を渡しておく。
ダウンロードであれば ServletOutputStream などを渡しておくとよい。

  - ビジネスロジック実装例

```
@Autowired
protected QueryRowHandleDAO queryRowHandleDAO = null;

    public int execute(BLogicParam arg0) {
        Parameter param = new Parameter();
        SampleRowHandler dataRowHandler = new SampleRowHandler();

        queryRowHandleDAO.executeWithRowHandler(
                "selectDataSql", param, dataRowHandler);

        // 終了コードの返却
        return 0;
    }
...(以下略)
```

実際に一件ずつ処理を行う DataRowHandler インスタンスを渡す。

※ TERASOLUNA Batch Framework for Java ver 3.x においてデータベースアクセス機能を使用する場合の注意点。
先の実装例にも掲載した通り、「CB-01 データベースアクセス機能」と異なり、ビジネスロジックの Bean 定義を行う必要はない。
TERASOLUNA Batch Framework for Java ver 3.x では、アノテーション(@Autowired や@Component)を利用してビジネスロジックと DAO の DI を行う。

### ◆ 拡張ポイント

なし。

## ■ 構成クラス

| | クラス名 | 概要 |
|---|---|---|
| 1 | jp.terasoluna.fw.dao.QueryDAO | 参照系 SQL を実行するための DAO インタフェース |
| 2 | jp.terasoluna.fw.dao.UpdateDAO | 更新系 SQL を実行するための DAO インタフェース |
| 3 | jp.terasoluna.fw.dao.StoredProcedureDAO | StoredProcedure を実行するための DAO インタフェース |
| 4 | jp.terasoluna.fw.dao.QueryRowHandleDAO | 参照系 SQL を実行し一件ずつ処理するための DAO インタフェース |
| 5 | jp.terasoluna.fw.dao.ibatis.QueryDAOiBatisImpl | QueryDAO インタフェースの iBATIS 用実装クラス |
| 6 | jp.terasoluna.fw.dao.ibatis.UpdateDAOiBatisImpl | UpdateDAO インタフェースの iBATIS 用実装クラス |
| 7 | jp.terasoluna.fw.dao.ibatis.StoredProcedureDAOiBatisImpl | StoredProcedureDAO インタフェースの iBATIS 用実装クラス |
| 8 | jp.terasoluna.fw.dao.ibatis.QueryRowHandleDAOiBatisImpl | QueryRowHandleDAO インタフェースの iBATIS 用実装クラス |

## ■ 関連機能

- なし

## ■ 使用例

- 機能網羅サンプル(terasoluna-batch-functionsample)
- チュートリアル(terasoluna-batch-tutorial)

# BL-07 ファイルアクセス機能

## ■ 概要

### ◆ 機能概要

- TERASOLUNA Batch Framework for Java ver 3.x で使用するファイルアクセス機能は、TERASOLUNA Batch Framework for Java ver 2.x で使用していたファイルアクセス機能と同一のものを利用して、ファイルアクセスを行う。
- 本項目では、TERASOLUNA Batch Framework for Java ver 3.x でファイルアクセス機能を使用する場合の TERASOLUNA Batch Framework for Java ver 2.x との違いのみを説明するものとし、ファイルアクセス機能の詳細な説明は別資料の「BC-01 ファイルアクセス機能」の機能説明書を参照すること。

### ◆ 概念図の違いについて

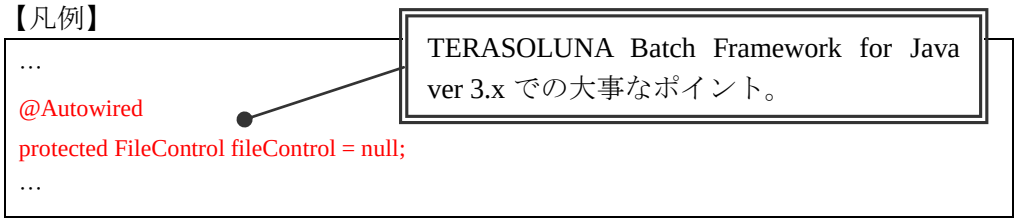
- 別資料の「BC-01 ファイルアクセス機能」では概念図「ファイル入力処理の概念図(Collector から利用される場合)」において Collector から利用される場合の概念図を掲載しているが、TERASOLUNA Batch Framework for Java ver 3.x では、このような使い方はしないため、読み飛ばすこと。
- TERASOLUNA Batch Framework for Java ver 3.x でファイルアクセス機能を利用するときはビジネスロジックから利用するので、「ファイル入力処理の概念図(ビジネスロジックから利用される場合)」の概念図を参考にすること。

## ◆ コーディングポイント

- 本説明書でのコーディングポイントは、別資料の「BC-01 ファイルアクセス機能」のコーディングポイントと異なる以下の項目についてのみ説明を行う。
  - ・ ファイル入力チェックについて
  - ・ 例外処理
  - ・ ファイル入力の実装例
  - ・ ファイル出力の実装例

  その他の項目については、別資料の「BC-01 ファイルアクセス機能」を参照すること。

- 次ページからのコーディングポイントの中で、TERASOLUNA Batch Framework for Java ver 3.x で本機能を使用する際に大事なポイントについては、二重線の吹き出しを使用して強調している。




- ファイル入力チェックについて

  TERASOLUNA Batch Framework for Java ver 3.x ではファイル、DB から取得したデータの入力チェックを行うための「AL-043 入力値検証機能」を提供している。
  使用方法についての詳細は「AL-043 入力値検証機能」の機能説明書を参照すること。

- 例外処理について

  TERASOLUNA Batch Framework for Java ver 3.x では例外処理のための「BL-04 例外ハンドリング機能」を提供している。
  使用方法についての詳細は「BL-04 例外ハンドリング機能」の機能説明書を参照すること。

- ファイル入力の実装例
  - ファイル入力処理の実装

(1) ファイル行オブジェクトを実装する。

(2) ファイル入力処理を行うクラスの Bean 定義を行う。
【ジョブ Bean 定義ファイルの設定例】

```
...
    <!-- コンポーネントスキャン設定 -->
    <context:component-scan base-package="jp.terasoluna.batch.sample.sample00001" />
...
```

Batch 3.x では@Component アノテーションを使用して、Bean を自動的に検出し、DI コンテナで管理する。

上記の例ではパッケージ[jp.terasoluna.batch.sample.sample00001]内に存在する@Component アノテーションが付与されたクラスを自動的に検出する。

(3) ファイル入力処理を行うクラスでは、FileQueryDAO の execute()メソッドでファイル入力用イテレータを取得する。ファイル入力用イテレータ取得時に、ファイルオープンが行われる。
ファイル入力用イテレータの next メソッドで、ファイル行オブジェクトを取得する。

➢ ビジネスロジック実装例

```
@Component
public class B001001BLogic implements BLogic {

    @Autowired
    @Qualifier("csvFileQueryDAO")
    protected FileQueryDAO fileQueryDAO = null;

    public int execute(BLogicParam arg0) {

        ...
        // ファイル入力用イテレータの取得
        FileLineIterator<SampleFileLineObject> fileLineIterator
                                = fileQueryDAO.execute(basePath +
                                        "/some_file_path/uriage.csv",FileColumnSample.class);

        try {

      // ヘッダ部の読み込み
          List<String> headerData = fileLineIterator.getHeader();
           ... // 読み込んだヘッダ部に対する処理

           while(fileLineIterator.hasNext()){
               // データ部の読み込み
             SampleFileLineObject sampleFileLine
                                        = fileLineIterator.next();

           ... // 読み込んだ行に対する処理
           }
          // トレイラ部の読み込み
          List<String> trailerData = fileLineIterator.getTrailer();
           ... // 読み込んだトレイラ部に対する処理

        } finally {
            // ファイルのクローズ
            fileLineIterator.closeFile();
        }
... (以下略)
```

Batch 3.x では@Autowired アノテーションを使用して、ビジネスロジックへの DAO の DI を行う。(FileDAO の場合は FileQueryDAO インターフェースの実装クラスが多数存在するので、@Qualifier を使用して Bean 名を指定する必要がある。)

ファイルパスとファイル行オブジェクトクラスを引数にして、ファイル入力用イテレータを取得する

アノテーション FileFormt の headerLineCount で設定した行数分のヘッダ部を取得する。

ファイル形式に関わらず、next()メソッドを使用する

アノテーション FileFormt の trailerLineCount で設定した行数分のトレイラ部を取得する

closeFile()メソッドでファイルを閉じること

➢ ファイルの入力順序
トレイラ部の入力は、データ部の入力が全て終わった後に行う必要がある点に留意すること。

➢ スキップ処理
ファイル入力機能では入力を開始する行を指定する事ができる。
スキップ処理は中断していたファイルの読み込みを再開するような場合に使用する事が出来る。

➢ ビジネスロジック実装例

```
……
// スキップ処理
fileLineIterator.skip(1000);
……
```

fileLineIterator のカレント行から 1000 行分のデータ行を読み飛ばす処理を行う

- ファイル出力の実装例
  - ファイルの設定として、囲み文字と区切り文字を設定し、データの一部をデフォルトのパディング文字でパディング処理したデータをファイルに出力する場合の記述例 (getter/setter は省略)

```
@FileFormat(delimiter = ",", encloseChar = '"' )
public class SampleFileLineObject {
     ……
    @OutputFileColumn(
           columnIndex = 0,
           columnFormat="yyyy/MM/dd")
    private Date hiduke = null;

    @OutputFileColumn(
           columnIndex = 1,
           paddingType = PaddingType.LEFT,
           bytes = 10,
           stringConverter = StringConverterToLowerCase.class)
    private String shopId = null;

    @OutputFileColumn(
           columnIndex = 2,
           columnFormat="###,###,###")
    private BigDecimal uriage = null;
     ……
}
```

アノテーションの FileFormat は必須
区切り文字、囲み文字を設定。

アノテーション OutputFileColumn
とパラメータの設定

パディング処理を行う場合は、バイト数の設定が必須となる。

    - 出力対象となるファイル行オブジェクトの値

```
hiduke = Sat Jul 01 00:00:00 JST 2006
shopId = SHOP01
uriage = 1000000
```

    - 上記のファイル行オブジェクトを出力すると以下の値となる。

```
"2006/07/01","    shop01","1,000,000"
```

- データの一部を個別のパディング文字でパディング処理したデータをファイルに出力する場合の記述例 (getter/setter は省略)

```
@FileFormat(delimiter = ",", encloseChar = '"' )
public class SampleFileLineObject {
     ……
    @OutputFileColumn(
           columnIndex = 0,
           columnFormat="yyyy/MM/dd")
    private Date hiduke = null;

    @OutputFileColumn(
           columnIndex = 1,
           paddingType = PaddingType.RIGHT,
           paddingChar = '0',
           bytes = 10,
           stringConverter = StringConverterToLowerCase.class)
    private String shopId = null;

    @OutputFileColumn(
           columnIndex = 2,
           columnFormat="###,###,###")
    private BigDecimal uriage = null;
     ……
}
```

右側のパディング処理を行い、パディング文字として'0'を設定する。

✧ 出力対象となるファイル行オブジェクトの値

```
 hiduke = Sat Jul 01 00:00:00 JST 2006
shopId = SHOP01
uriage = 1000000
```

✧ 上記のファイル行オブジェクトを出力すると以下の値となる。

```
"2006/07/01","shop010000","1,000,000"
```

- ファイル出力処理の実装（１メソッドでファイルをオープン・クローズする場合）
-

(1) ファイル行オブジェクトを実装する。

(2) ファイル出力処理を行うクラスの Bean 定義を行う。

【ジョブ Bean 定義ファイルの設定例】

```
...
    <!-- コンポーネントスキャン設定 -->
    <context:component-scan base-package="jp.terasoluna.batch.sample.Sample00001" />
...
```

Batch 3.x では@Component アノテーションを使用して、Bean を自動的に検出し、DI コンテナで管理する。

上記の例ではパッケージ[jp.terasoluna.batch.sample.Sample00001]内に存在する@Component アノテーションが付与されたクラスを自動的に検出する。

※ファイル入力処理の Bean 定義の際に上記の記述を行っていた場合は、改めて記述する必要はない。

(3) ファイル出力処理を行うクラスでは、FileUpdateDAO の execute メソッドでファイル出力用行ライタを取得する。ファイル出力用行ライタの取得時に、ファイルがオープンされる。

➢ ビジネスロジック実装例

```
@Component
public class B001001BLogic implements BLogic {

    @Autowired
    @Qualifier("csvFileUpdateDAO")
    protected FileUpdateDAO csvFileUpdateDAO = null;

    public int execute(BLogicParam arg0) {

        // ファイル出力用行ライタの取得
        FileLineWriter< SampleFileLineObject > fileLineWriter
                = fileUpdateDAO.execute(basePath + "/some_file_path/uriage.csv",
                                        SampleFileLineObject.class);

        try {
            // ヘッダ部の出力
            fileLineWriter.printHeaderLine(headerString);
            ...
        while ( ... ) {
            ...
            // データ部の出力（1行）
            fileLineWriter.printDataLine(sampleFileLineObject);
            ...
        }
        ...
        // トレイラ部の出力
        fileLineWriter.printTrailerLine(trailerString);
        ...
    } finally {
        // ファイルのクローズ
        fileLineWriter.closeFile();
    }
...(以下略)
```

ファイル入力の際と同様に @Autowired アノテーションを使用して、ビジネスロジックへの DAO の DI を行う。

ファイル名とパラメータクラスを引数に、ファイル出力用行ライタを取得する。

ヘッダ部を出力する、String 型の変数を引数とする。

ファイル形式に関わらず、printDataLine メソッドで出力する。
出力される項目には、項目定義用のアノテーションを付加しておく。

トレイラ部を出力する。String 型の変数を引数とする。

出力が終了したら、ファイルをクローズする。

### ◆ 拡張ポイント

- なし

## ■ 関連機能

- 『AL-043 入力値検証機能』
- 『BL-04 例外ハンドリング機能』

## ■ 使用例

- 機能網羅サンプル(terasoluna-batch-functionsample)
- チュートリアル(terasoluna-batch-tutorial)

## ■ 備考

- なし

# BL-08 ファイル操作機能

## ■ 概要

### ◆ 機能概要

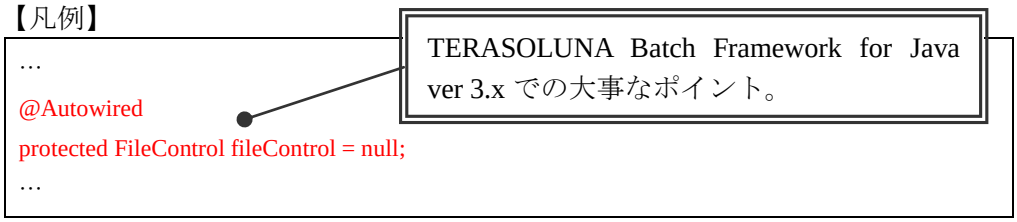
- TERASOLUNA Batch Framework for Java ver 3.x で使用するファイル操作機能は、TERASOLUNA Batch Framework for Java ver 2.x で使用していたファイル操作機能と同一のものを利用して、ファイル操作を行う。
- 本項目では、TERASOLUNA Batch Framework for Java ver 3.x でファイル操作機能を使用する場合の TERASOLUNA Batch Framework for Java ver 2.x との違いのみを説明するものとし、ファイル操作機能の詳細な説明は別資料の「BC-02　ファイル操作機能」の機能説明書を参照すること。

### ◆ コーディングポイント

- 本説明書でのコーディングポイントは、別資料の「BC-02　ファイル操作機能」のコーディングポイントと異なる以下の項目についてのみ説明を行う。
  - ・　ファイル操作クラス（FileControl）を利用する例

  その他の項目については、別資料の「BC-02　ファイル操作機能」を参照すること。

- 次ページからのコーディングポイントの中で、TERASOLUNA Batch Framework for Java ver 3.x で本機能を使用する際に大事なポイントについては、二重線の吹き出しを使用して強調している。



- ファイル操作クラス（FileControl）を利用する例
  - Bean 定義ファイル(commonContext.xml)の設定例

```
……
<bean id="fileControl"
      class="jp.terasoluna.fw.file.util.FileControlImpl">
      <property name="basePath" value="${basepath}" />
      <property name="checkFileExist" value="false" />
</bean>
……
```

FileControl インタフェースを実装するクラスをフレームワーク Bean 定義ファイルに定義する。
プロパティに基準パスを設定すること。

操作後にできるファイルパスにファイルが存在する場合、処理を継続する(上書きする:true)か例外を投げて停止する(false)かを決めるフラグ。

  - ビジネスロジックの実装例（ファイルのコピー、移動、削除処理の実装例）

```
@Component
public class Sample001BLogic implements BLogic {
      ……
      @Autowired
      protected FileControl fileControl = null;
      (…中略…)
      public int execute(BLogicParam arg0) {
            // ファイルのコピー（相対パスを設定する例）
            //   /si1/chohyo/testtxt を/si1/chohyo/testFile.txt にコピー。
            // 基準パスは「/si1/」
            fileControl.copyFile("chohyo/test.txt", "chohyo/testFile.txt");
            ……
            // ファイルの移動（相対パスを設定する例）
            //   /si1/chohyo/testFile.txt を/si1/output/testFile.txt に移動。
            // 基準パスは「/si1/」
            fileControl.renameFile("chohyo/testFile.txt", "output/testFile.txt");
            ……
            //ファイルの削除（相対パスを設定する例）
            //   /si1/chohyo/testFile.txt を削除。
            //基準パスは「/si1/」
            fileControl.deleteFile("chohyo/testFile.txt");
            ……
            // ファイルのコピー（絶対パスを設定する例）
            //   /si1/chohyo/test.txt を/si1/chohyo/testFile.txt にコピー。
            fileControl.copyFile("/si1/chohyo/test.txt", "/si1/chohyo/testFile.txt");
            ……
(…以下略…)
```

ファイル操作機能を利用するビジネスロジックは、@Autowired を使用して FileControl インタフェース実装クラスをビジネスロジックに DI する。

各メソッドの引数はファイルの相対パス、もしくは絶対パスを記述する

➢ ビジネスロジックの実装例（ファイル結合の実装例）

```
……
// ファイルの結合。
// 以下に挙げるファイルをリストに格納し、ファイルを/si1/output/mergeFile.csv に統合。
//   /si1/chohyo/output001.csv
//   /si1/chohyo/output002.csv
//   /si1/chohyo/output003.csv
// 基準パスは「/si1/」
fileList.add("chohyo/output001.csv");
fileList.add("chohyo/output002.csv");
fileList.add("chohyo/output003.csv");
……
fileControl.mergeFile(fileList, "output/mergeFile.csv");
……
```

メソッドの 2 番目の引数はファイルの相対パス、もしくは絶対パスを記述する

- ファイル操作ユーティリティクラス（FileUtility）を直接利用する例

➢ ビジネスロジックの実装例（ファイルのコピー、移動、削除処理の実装例）

```
……
// ファイルのコピー。
//   /si1/chohyo/test.txt を/si1/chohyo/testFile.txt にコピー。
FileUtility.copyFile("/si1/chohyo/test.txt", "/si1/chohyo/testFile.txt");
……
// ファイルの移動。
//   /si1/chohyo/testFile.txt を/si1/output/testFile.txt に移動。
FileUtility.renameFile("/si1/chohyo/testFile.txt",
      "/si1/output/testFile.txt");
……
//ファイルの削除。    /si1/chohyo/testFile.txt を削除。
FileUtility.deleteFile("/si1/chohyo/testFile.txt");
……
```

各メソッドの引数はファイルの絶対パスを記述する

## ◆ 拡張ポイント

- なし。

# BL-09 メッセージ管理機能

## ■ 概要

### ◆ 機能概要

- TERASOLUNA Batch Framework for Java ver 3.x で使用するメッセージ管理機能は、TERASOLUNA Server Framework for Java ver 2.x で使用していたデータベースアクセス機能と同一のものを利用して、メッセージ管理を行う。
- 本項目では、TERASOLUNA Batch Framework for Java ver 3.x でデータベースアクセス機能を使用する場合の TERASOLUNA Server Framework for Java ver 2.x との違いのみを説明するものとし、メッセージ管理機能の詳細な説明は別資料の「CE-01 メッセージ管理機能」の機能説明書を参照すること。

## ◆ コーディングポイント

- 本説明書でのコーディングポイントは、別資料の「CE-01 メッセージ管理機能」のコーディングポイントと異なる以下の項目についてのみ説明を行う。
  - ・ ソフトウェアアーキテクトが行うコーディングポイント（リソースバンドル）
  - ・ ソフトウェアアーキテクトが行うコーディングポイント（ＤＢメッセージ)
  - ・ 業務開発者が行うコーディングポイント

  その他の項目については、「CE-01 メッセージ管理機能」を参照すること。

- ソフトウェアアーキテクトが行うコーディングポイント（リソースバンドル）
  以下のように "messageSource"という識別子の Bean を準備することで、この機能を利用できる。
  - ➢ Bean 定義ファイルサンプル（commonContext.xml）

messageSource を指定する。

```
<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
       <property name="basenames" value="application-messages, system-messages"/>
</bean>
```

読み込むプロパティファイルをカンマ区切りで列挙する。
ファイル名の".properties"は省略する。

プロパティファイルはクラスパス上に配置する。

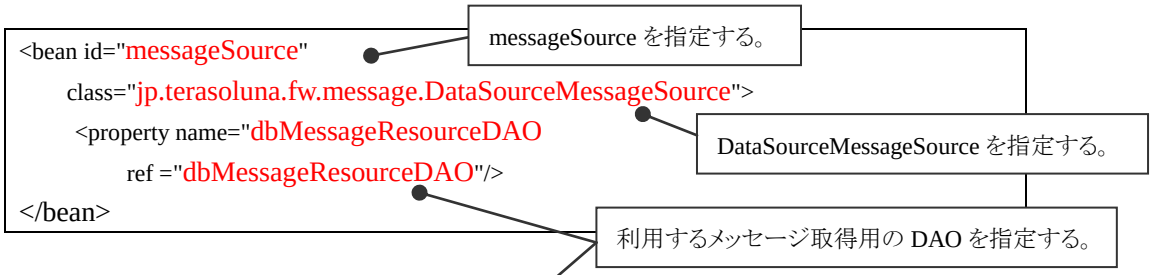
定義するプロパティファイルが多い場合は、下記のようにリストの形で指定することもできる。

- ➢ Bean 定義ファイルサンプル（commonContext.xml）

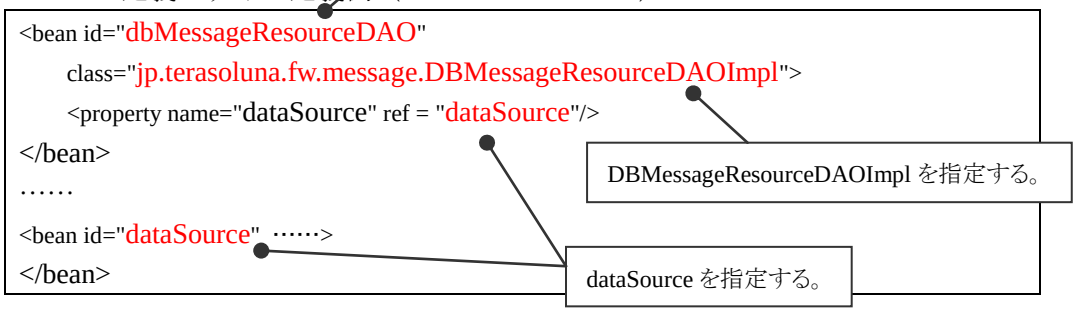
```
<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
      <property name="basenames">
            <list>
                  <value> application-messages </value>
                  <value>system-messages</value>
            </list>
      </property>
</bean>
```

- ソフトウェアアーキテクトが行うコーディングポイント（ＤＢメッセージ)
  以下のように "messageSource"という識別子の Bean を準備することで、この機能を利用できる。

  - メッセージリソース取得 Bean の定義
    TERASOLUNA Server Framework for Java が提供している DataSourceMessageSource クラスを指定し、DAO(後述)を DI する。BeanID は”messageSource”である必要がある。
  - Bean 定義ファイル定義例（commonContext.xml）




  - メッセージリソース取得用 DAO の Bean 定義
    DBMessageResourcesDAO インタフェースを指定し、データソースを DI する。TERASOLUNA Server Framework for Java ではこのインタフェースのデフォルト実装として DBMessageRecourceDAOImpl クラスを提供している。
  - Bean 定義ファイル定義例（commonContext.xml）




  - データソースの定義
    『データベースアクセス機能』を参照して設定する。
  - メッセージ文字列の定義

メッセージ文字列はデータベース中の以下のテーブルに格納しておく:
・テーブル名 ： MESSAGES
・メッセージコードを格納するカラム名 ： CODE
・メッセージ本文を格納するカラム名 ： MESSAGE
DBMessageResourceDAOImpl が発行する SQL は以下である。

```
SELECT CODE,MESSAGE FROM MESSAGES
```

テーブルスキーマを自由に定義することも可能である。後述「メッセージリソースのテーブルスキーマをデフォルト値から変更する」を参照されたい。

- 業務開発者が行うコーディングポイント
  - TERASOLUNA Batch Framework for Java ver 3.x でのメッセージの取得方法
    - ビジネスロジック実装例

```
public class SampleBLogic implements BLogic {

    //ビジネスロジック
    public int execute (BLogicParam arg0) {
        …中略…

        } catch(RuntimeException e) {
            if(log.isErrorEnabled()){
                log.error(MessageUtil.getMessage("errors.runtimeexception"));
                return 255;
            }
        }

        …中略…
    }
}
```

MessageUtil クラスのメソッドを利用して、メッセージを取得する。

  - メッセージリソースの再定義方法

    Web アプリケーション起動中にアプリケーションコンテキスト内のメッセージをデータベースから再取得することができる。再定義には以下のメソッドを使用する。なお、クラスタ環境化では、クラスタごとに再定義する必要があるので注意されたい。以下のメソッドを使用する。

    **jp.terasoluna.fw.message.DataSourceMessageSource クラスの reloadDataSourceMessage メソッド**

    各ビジネスロジックが直接、reloadDataSourceMessage メソッドを使用することはせず、業務共通クラスから利用することを推奨する。

# ■ メッセージの国際化対応

- ソフトウェアアーキテクトが行うコーディングポイント
  - ➢ デフォルトロケールの設定
    メッセージリクエストにロケールが設定されていない場合、及びメッセージリソース内にメッセージリクエストで要求されたロケールが見つからない場合に使用される。設定しない場合はデフォルトロケールの初期設定（サーバー側 VM のロケール）が使用される。
  - ➢ Bean 定義ファイル定義例（commonContext.xml）

```
<bean id="messageSource"
      class="jp.terasoluna.fw.message.DataSourceMessageSource">
      <property name="dbMessageResourceDAO"
            ref ="dbMessageResourceDAO"/>
      <property name="defaultLocale" value="ja_JP"/>
</bean>
```

デフォルトロケールを指定する。

  - ➢ 国際化対応カラムの有効化
    データベースのロケールに対応するカラムからの読込みを有効にする必要がある。ロケールに対応するカラムは以下の３つがある。

    ・言語コードカラム
    ・国コードカラム
    ・バリアントコードカラム

    設定の優先順位は、言語コードカラムが一番高く、国コードカラム、バリアントコードカラムの順に低くなる。言語コードカラムを指定せずに、国コードカラムやバリアントコードカラムを指定しても無効となる。

    これらのカラムのうち、言語コードカラムの指定によってデータベースに登録されたメッセージの認識が以下のように変化する。

    - **言語コードカラムを指定しない場合**は、すべてのメッセージがデフォルトロケールとして認識される。（defaultLocale プロパティを指定した場合はその値となる）
    - **言語コードカラムを指定した場合**は、言語コードカラムに指定したとおりに認識される。

    注意点としては、**言語コードカラムを指定し、言語コードカラムに null や空文字のメッセージをデータベースに登録した場合**、そのメッセージは**アプリケーションから参照されない**点である。null や空文字で登録したメッセージがデフォルトロケールとして認識されるわけではない点に注意。

以下のプロパティで設定されていない値はデフォルトの値が使用される。設定する項目は以下の通り。

| プロパティ名 | デフォルト値 | 概要 | 備考 |
|---|---|---|---|
| languageColumn | null | 言語コードを格納するカラム名 | 国際化対応時のみ設定 |
| countryColumn | null | 国コードを格納するカラム名 | 国際化対応時のみ設定 |
| variantColumn | null | バリアントコードを格納するカラム名 | 国際化対応時のみ設定 |

メッセージ取得 SQL 文のフォーマットは以下の通り。

**SELECT メッセージコード, (言語コード), (国コード), (バリアントコード), メッセージ本体 FROM テーブル名 FROM テーブル名**

()内は設定した値のみが有効になる。デフォルトでは無効になっており、カラム名を設定すると有効になる。

- Bean 定義ファイル定義例（commonContext.xml）

```
<bean id=DBMessageResourceDAO
      class="jp.terasoluna.fw.message.DBMessageResourceDAOImpl">
      <property name="dataSource" ref bean="dataSource"/>
      <property name=tableName value="DBMESSAGES"/>
      <property name=codeColumn value="BANGOU"/>
      <property name=languageColumn value="GENGO"/>
      <property name=countryColumn value="KUNI</value>
      <property name=variantColumn value="HOUGEN</value>
      <property name=messageColumn  value="HONBUN"/>
</bean>
```

国際化対応する場合のみ設定。
言語コードのカラム名を指定する。

国際化対応する場合のみ設定。
国コードのカラム名を指定する。

国際化対応する場合のみ設定。
バリアントコードのカラム名を指定する。

DBのテーブル名及びカラム名は以下の様な設定となる。
テーブル名 = DBMESSAGES
メッセージコードを格納するカラム名 = BANGOU
メッセージの言語コードを格納するカラム名 = GENGO
メッセージの国コードを格納するカラム名 = KUNI
メッセージのバリアントコードを格納するカラム名 = HOUGEN
メッセージ本文を格納するカラム名 = HONBUN
検索SQL文は以下の通り。
**SELECT BANGOU,GENGO,KUNI,HOUGEN,HONBUN FROM**

# ■ リファレンス

## ◆ 構成クラス

| | クラス名 | 概要 |
|---|---|---|
| 1 | DataSourceMessageSource | メッセージを生成、発行するクラス |
| 2 | DBMessageResourceDAOImpl | DB よりメッセージリソースを抽出する DBMessageResourceDAO の実装クラス |
| 3 | MessageSource | メッセージ取得のためのメソッドを規定したインタフェイスクラス |

## ◆ 拡張ポイント

- なし

# ■ 関連機能

- なし

# ■ 使用例

- 機能網羅サンプル(terasoluna-batch-functionsample)

# ■ 備考

## ◆ メッセージリソースのテーブルスキーマをデフォルト値から変更する

- テーブル名、カラム名をプロジェクト側で独自に指定する場合
  テーブル名及び各カラム名のすべてもしくは一部を設定することでデータベースのテーブル名及びカラム名を自由に変更できる。設定されていない値はデフォルトの値が使用される。設定する項目は以下の通り。

| プロパティ名 | デフォルト値 | 概要 |
|---|---|---|
| tableName | MESSAGES | テーブル名 |
| codeColumn | CODE | メッセージコードを格納するカラム名 |
| messageColumn | MESSAGE | メッセージ本文を格納するカラム名 |

メッセージ取得 SQL 文の SELECT 節のフォーマットは以下の通り。

**SELECT メッセージコード, メッセージ本体**

なお、この設定では国際化に未対応となる。国際化対応が必要な場合は、前述の『メッセージの国際化対応』の項目を参照のこと。

例）データベースのテーブル名及びカラム名を以下の様にする場合
・テーブル名 ： DBMESSAGES
・メッセージコードを格納するカラム名 ： BANGOU
・メッセージ本文を格納するカラム名 ： HONBUN

➢ Bean 定義ファイル定義例（commonContext.xml）

```
<bean id="dBMessageResourceDAO"
      class="jp.terasoluna.fw.message.DBMessageResourceDAOImpl">
      <property name="dataSource" ref bean="dataSource"/>
      <property name="tableName" value="DBMESSAGES"/>
      <property name="codeColumn" value="BANGOU"/>
      <property name="messageColumn" value="HONBUN"/>
</bean>
```

テーブル名を指定する。

メッセージコードのカラム名を指定する。

メッセージ本文のカラム名を指定する。

メッセージ取得 SQL 文は以下の通り。
**SELECT BANGOU,HONBUN FROM DBMESSAGES**

- 上記『テーブル名、カラム名をプロジェクト側で独自に指定する場合』に加え、プロジェクト独自の SQL 文を設定する場合
  findMessageSql プロパティで独自の SQL 文を設定できる。設定する SQL 文には、

codeColumn プロパティおよび messageColumn で指定したカラムが必要となる。
設定する項目は以下の通り。

| プロパティ名 | デフォルト値 | 概要 |
|---|---|---|
| tableName | MESSAGES | テーブル名 |
| codeColumn | CODE | メッセージコードを格納するカラム名 |
| messageColumn | MESSAGE | メッセージ本文を格納するカラム名 |
| findMessageSql | - | メッセージを取得する SQL 文 |

例）メッセージ取得 SQL 文を『SELECT BANGOU,HONBUN FROM DBMESSAGE WHERE CATEGORY='TERASOLUNA'』とする場合。
・テーブル名 ： DBMESSAGES
・メッセージコードを格納するカラム名 ： BANGOU
・メッセージ本文を格納するカラム名 ： HONBUN

➢ Bean 定義ファイル定義例（commonContext.xml）

```
<bean id ="dBMessageResourceDAO"
      class="jp.terasoluna.fw.message.DBMessageResourceDAOImpl">
      <property name="dataSource" ref bean="dataSource"/>
      <property name="tableName" value="DBMESSAGES"/>
      <property name="codeColumn" value="BANGOU"/>
      <property name="messageColumn" value="HONBUN"/>
      <property name="findMessageSql"
          value=
          "SELECT BANGOU,HONBUN FROM DBMESSAGE WHERE
          CATEGORY='TERASOLUNA'"
      />
</bean>
```

テーブル名を指定する。

メッセージコードのカラム名を指定する。

メッセージ本文のカラム名を指定する。

検索 SQL 文を指定する。

## ◆ 第2メッセージリソースの使用

メッセージリソースを追加できる。追加したメッセージリソースは前述で"messageSource"という識別子の Bean として設定したメッセージリソースでメッセージが決定できない場合に利用される。以下のように “parentMessageSource プロパティ”に別のメッセージリソースへの参照を指定することで、この機能を利用できる。

- 第２メッセージリソース取得 Bean の定義
  利用したいメッセージリソース取得クラスを指定する。 BeanID は”messageSource”とは別の名前を付与する必要がある。

AbstractMessageSource の継承クラスであれば、この“parentMessageSource プロパティ”を利用できるので、さらに第 3、4 とリンクすることが可能である。

- Bean 定義ファイル定義例（commonContext.xml）

```
<bean id="messageSource"
    class="jp.terasoluna.fw.message.DataSourceMessageSource">
    <property name="parentMessageSource" ref="secondMessageSource"/>
    <property name="dbMessageResourceDAO" ref="dbMessageResourceDAO"/>
</bean>

<bean id="secondMessageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basenames" value="applicationResources,errors"/>
</bean>
```

messageSource を指定する。
優先して検索されるメッセージリソースとなる。

次に参照される

第２のメッセージリソースを指定する。
上記 messageSource 内にメッセージが存在しなかった場合、ここで指定したメッセージリソース内を検索する。

# AL-036 バッチ更新最適化機能

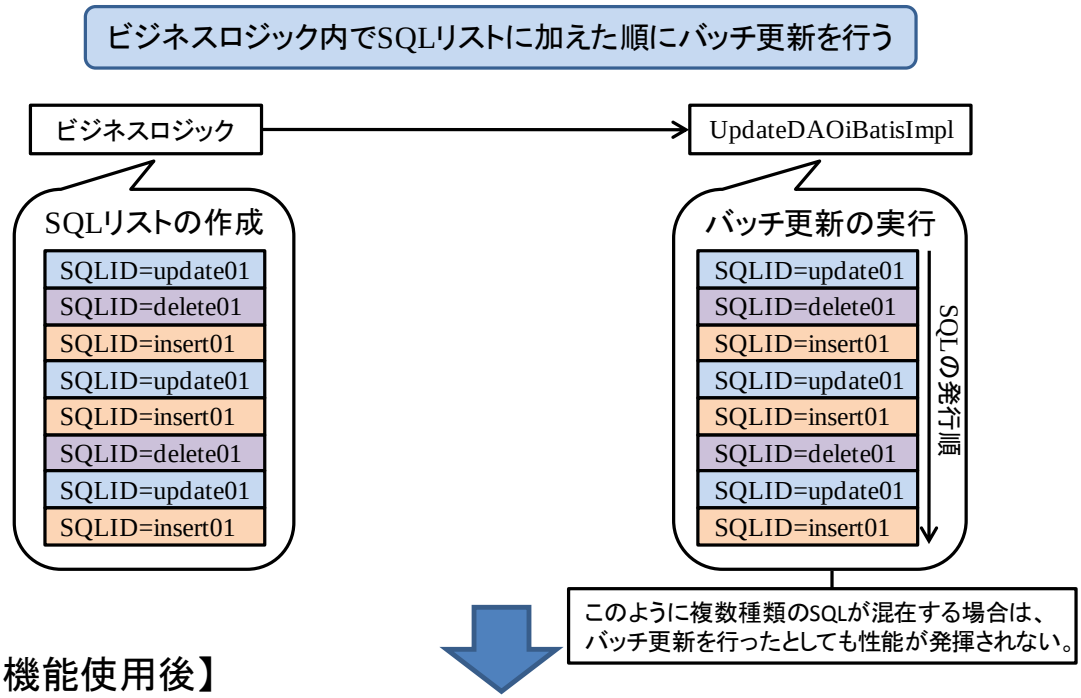
## ■ 概要

### ◆ 機能概要

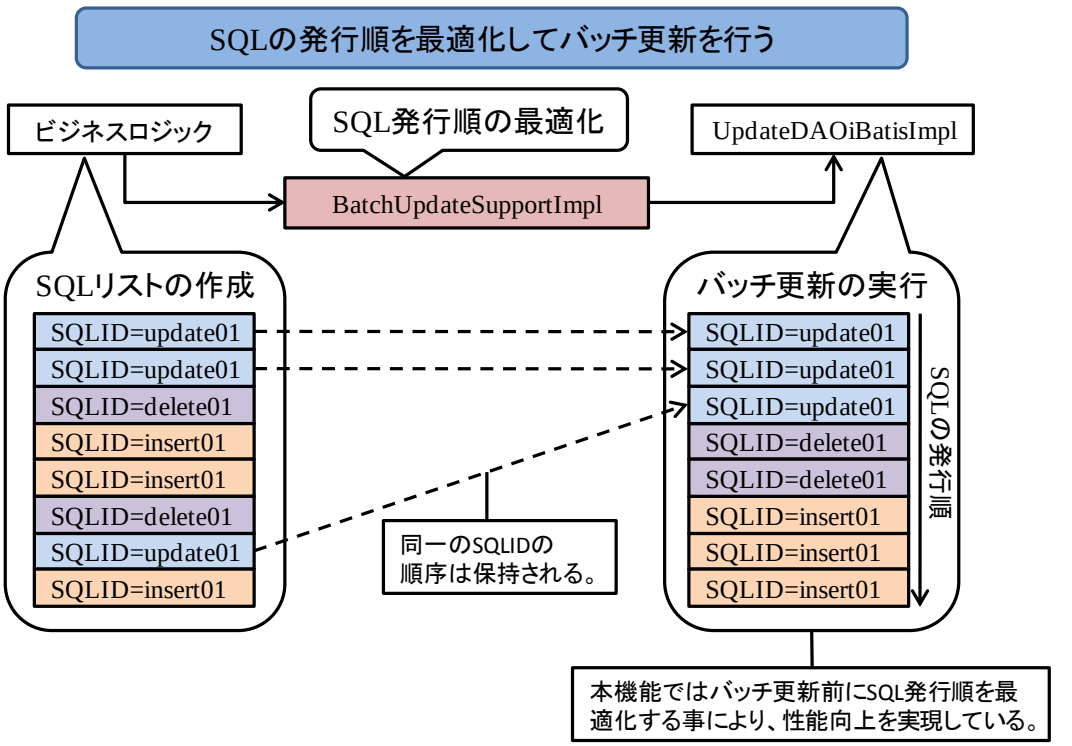
- バッチ更新を行う前に SQL の発行順を最適化する機能を提供する。
- SQL の最適化を行うと、バッチ更新時に発行する SQL の種類が 2 種類以上の場合に、本機能による性能の向上が期待できる。
  - （発行する SQL が単一の場合は、従来のバッチ更新との差は無い）
- 最適化により SQL の発行順が変更される為、「AL-036 バッチ更新最適化機能」を使用する際は、SQL の発行順が変更されても問題が無い事が前提条件となる。

## ◆ 概念図

【機能使用前】




【機能使用後】



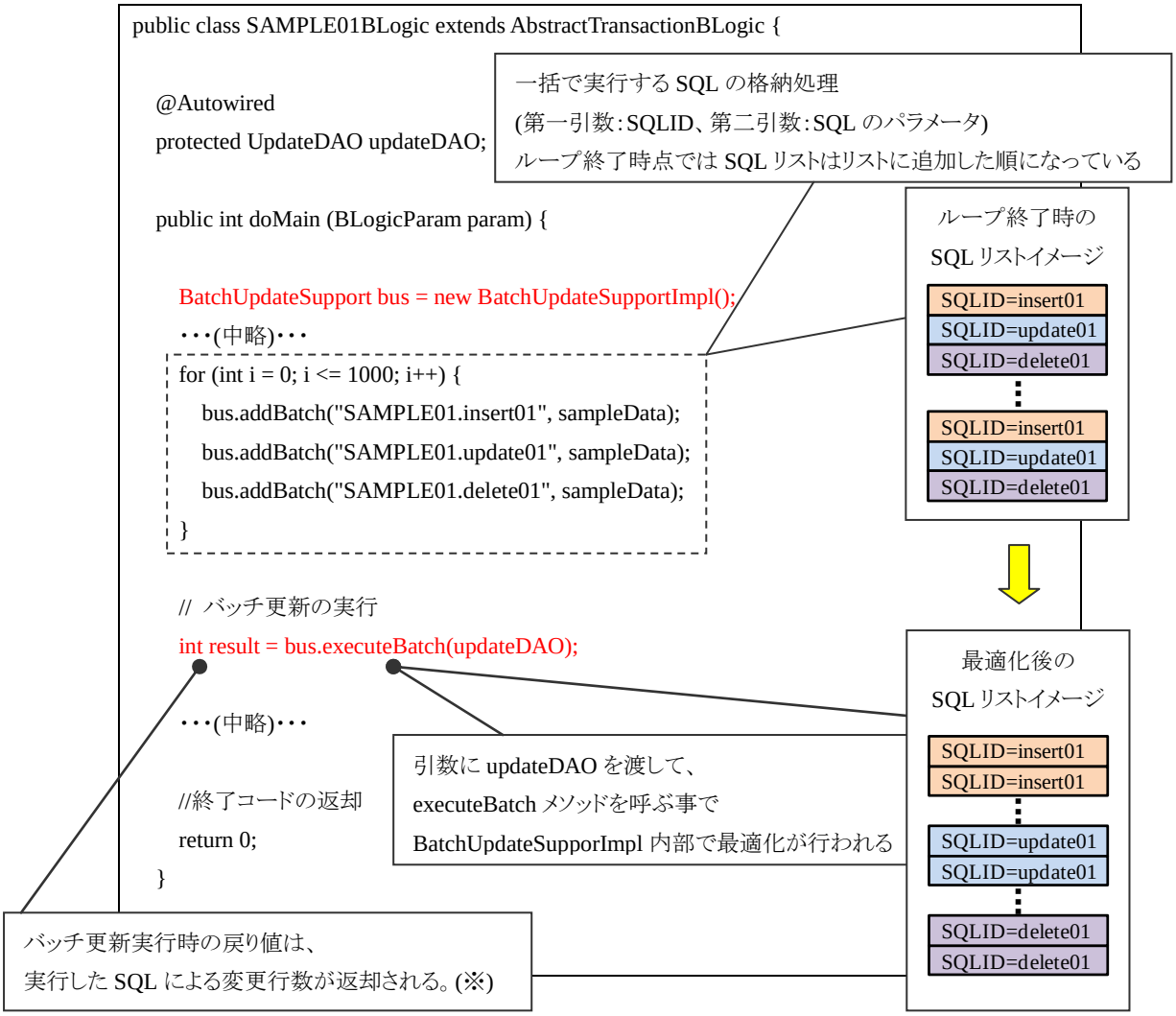
## ◆ 解説

- SQL 発行順の最適化のために、SQLID をキーにして並び変える。
  - SQL の発行順は、デフォルトではユニークな SQLID が SQL リストに登録された順を保持する。
  - 例えば「A,B,C,D」という４種の SQLID が「C,A,B,D,C,B,A」の順に SQL リストに格納されていたとすると、ユニークな SQLID の順は「C,A,B,D」となる。
    この順番を保持したまま最適化が行われるため、バッチ更新時の SQL 発行順は「C,C,A,A,B,B,D」の順となる。
  - ソート順は後述の方法により変更可能である(後述の sort メソッドを使用するか、拡張ポイントの項目を参照する事)
- 同一の SQLID 間では、概念図中の破線のように、最適化時に SQL リストに格納された順序を保持する。
- 最適化により、同一 SQL を連続して発行する事により、PreparedStatement オブジェクトを有効利用する事が出来る。
  その結果、PreparedStatement オブジェクトの生成回数の減少、通信回数の削減(※)につながり、性能の向上が期待できる。

  ※ PostgreSQL,OracleDatabase については通信回数の削減を確認している。

## ◆ コーディングポイント

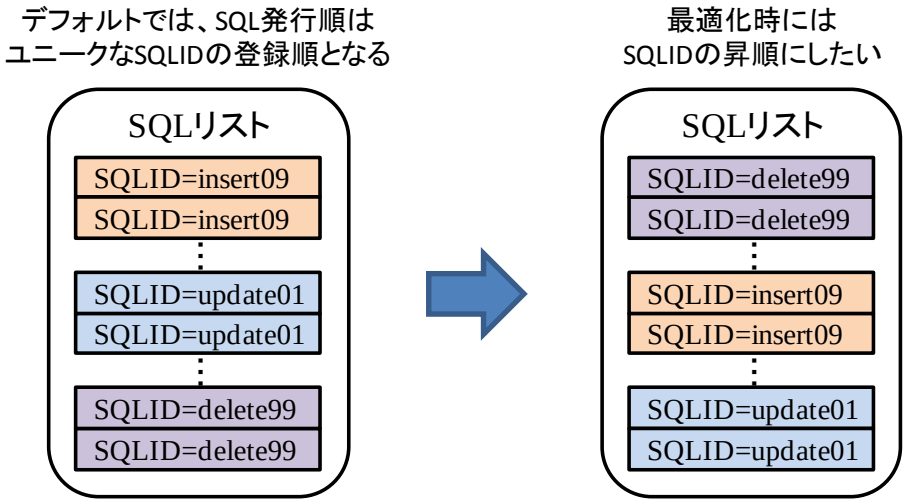
- 本機能を使用する際の実装例
  - ビジネスロジック実装例(TERASOLUNA Batch Framework for Java ver 3.x の場合)




※ java.sql.PreparedStatement を使用しているため、ドライバにより正確な行数が取得できないケースがある。
変更行数が正確に取得できないドライバを使用する場合や、変更行数がトランザクションに影響を与える業務(変更行数が 0 件の場合エラー処理をするケース等)では、バッチ更新は使用しないこと。
(参考資料)
http://otndnld.oracle.co.jp/document/products/oracle10g/101/doc_v5/java.101/B13514-02.pdf
450 ページ「標準バッチ処理の Oracle 実装の更新件数」を参照のこと。

- SQL の発行順序に関する注意点
  - 最適化により、SQL の発行順が変更されてしまうため、SQL の発行順に意味があるような場合は、発行順が保持されるように注意する事。

- sort メソッドを使用して昇順に最適化を行う
  - BatchUpdateSupportImpl クラスが持つ sort メソッドを使用する事により、最適化後の SQL 発行順を昇順に並び変える事が出来る。
  - ビジネスロジック中での sort メソッドの使用例を以下に例を挙げて掲載する。




  - ビジネスロジッククラスの実装例(部分的に抜粋)

```
・・・
BatchUpdateSupport bus = new BatchUpdateSupportImpl();
・・・(中略) ・・・
// SQL 発行順のソート
bus.sort()
// バッチ更新の実行
bus.executeBatch(updateDAO);
・・・
```

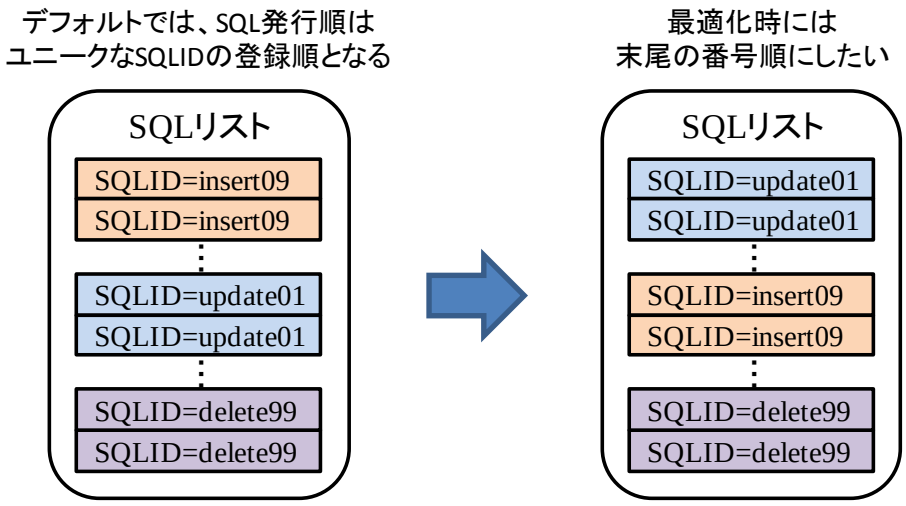
バッチ更新実行前に、sort メソッドを呼び出す事によって、内部的にソートフラグが立つ。

実際にソートが行われるのはバッチ更新実行時。

  - このように実装する事により、バッチ更新実行時に SQL の発行順が SQLID の昇順となるようにソートされる。
  - また、sort メソッドは引数として java.util.Comparator インタフェースの実装クラスを渡す事により、昇順以外の順に並び変える事も可能である。
    （詳細は拡張ポイントの項目を参照する事）

## ◆ 拡張ポイント

- SQL 最適化時のソート順を変更する方法（sort メソッドを使用する方法）
  java.util.Comparator インタフェースの実装クラスを作成し、バッチ更新実行前に sort メソッドを呼び出す事によって、ソート順を変更する事ができる。
  - ➢ 以下に例を挙げて「java.util.Comparator インタフェースの実装クラス」とビジネスロジックの実装例を掲載する。




➢ java.util.Comparator インタフェースの実装クラスの実装例

```
public class CustomSort implements Comparator<String> {

    public int compare(String str1, String str2) {
        if (str1 != null && str2 != null) {
            String subStr1 = str1.substring(str1.length() - 2);
            String subStr2 = str2.substring(str1.length() - 2);
            return subStr1.compareTo(subStr2);
        }
        return 0;
    }

}
```

Comparator インタフェースの実装
Compare メソッドを作成する。

～このロジックの簡単な説明～
渡された文字列の末尾 2 文字を取得し、比較して昇順になるようにしている。
引数のどちらかが null の場合は 0 を返す。

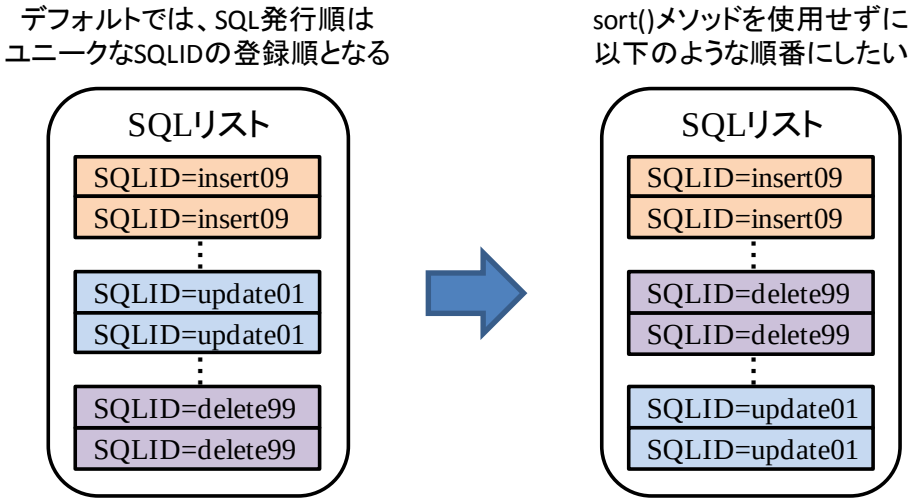
➢ ビジネスロジッククラスの実装例(部分的に抜粋)

```
・・・
BatchUpdateSupport bus = new BatchUpdateSupportImpl();
・・・(中略) ・・・
// SQL 発行順のソート
bus.sort(new CustomSort())
// バッチ更新の実行
bus.executeBatch(updateDAO);
・・・
```

作成した java.util.Comparator インタフェース実装クラスのインスタンスを生成し、sort メソッドの引数として渡す。

実際にソートが行われるのは
バッチ更新実行時。

- SQL 最適化時のソート順を変更する方法（直接 SQLID を指定する方法）
  バッチ更新時に SQLID を直接指定する方法でも、SQL の発行順を変更する事が可能である。
  - 以下に例を挙げて SQLID を直接指定して、SQL の発行順を変更する際のビジネスロジックの実装例を掲載する。




  - ビジネスロジッククラスの実装例(部分的に抜粋)

```
・・・
BatchUpdateSupport bus = new BatchUpdateSupportImpl();
・・・(中略) ・・・
// バッチ更新の実行
bus.executeBatch(updateDAO,
      "Common.insertData09",
      "Common.deleteData99",
      "Common.updateData01");
・・・
```

バッチ更新実行時に、SQLID を文字列で直接指定する。

  - 直接 SQLID を指定する方法を使用する場合は、ビジネスロジック中で SQL リストに格納される可能性のある全ての SQLID を指定しておく必要がある。
  - 以下のようにバッチ更新実行時に、リストに格納されている SQLID の指定がされていなかった場合は、バッチ更新実行時にエラーコード-200 が返却される。
    - ビジネスロジッククラスの実装例(部分的に抜粋)

```
・・・
BatchUpdateSupport bus = new BatchUpdateSupportImpl();
・・・(中略) ・・・
// バッチ更新の実行
bus.executeBatch(updateDAO,
      "Common.insertData09",
      "Common.deleteData99");
・・・
```

SQL リストに格納されている
”Commom.updateData01”を指定していないので
バッチ実行時にエラーコード-200 が返却される。

# ■ リファレンス

## ◆ 構成クラス

| | クラス名 | 概要 |
|---|---|---|
| 1 | jp.terasoluna.fw.batch.dao.support.BatchUpdateExecutor | バッチ更新一括実行クラス。<br>オブジェクト内に含まれる複数のバッチ更新を一括で実行する。 |
| 2 | jp.terasoluna.fw.batch.dao.support.BatchUpdateResult | バッチ更新実行結果クラス。<br>BatchUpdateExecutor によるバッチ更新一括実行の結果として各々のバッチ更新の結果を、リストにまとめて返却される。 |
| 3 | jp.terasoluna.fw.batch.dao.support.BatchUpdateSupport | バッチ更新サポートインタフェース。<br>バッチ更新用の SQL 追加メソッドやバッチ更新実行メソッド等を定義している。 |
| 4 | jp.terasoluna.fw.batch.dao.support.BatchUpdateSupportImpl | バッチ更新サポートインタフェースの実装クラス。<br>SQL の最適化やバッチ更新を行う。 |

## ◆ 関連機能

- 『BB-01 データベースアクセス機能』

## ◆ 使用例

- **機能網羅サンプル**(terasoluna-batch-functionsample)

## ◆ 備考

- 動的 SQL の使用について
  - 本機能は SQLID の並べ替えによる最適化を行うものである。
  - 動的 SQL を使用した場合、同一の SQLID であっても渡されるパラメータにより、毎回異なる SQL が発行される可能性がある。
  - そのため、動的 SQL を使用した場合は静的 SQL を使用する場合と比べ、使用した動的 SQL の複雑さに因って性能向上効果は薄れてしまうと見込まれる。

# AL-041　入力データ取得機能

## ■ 概要

### ◆ 機能全体像

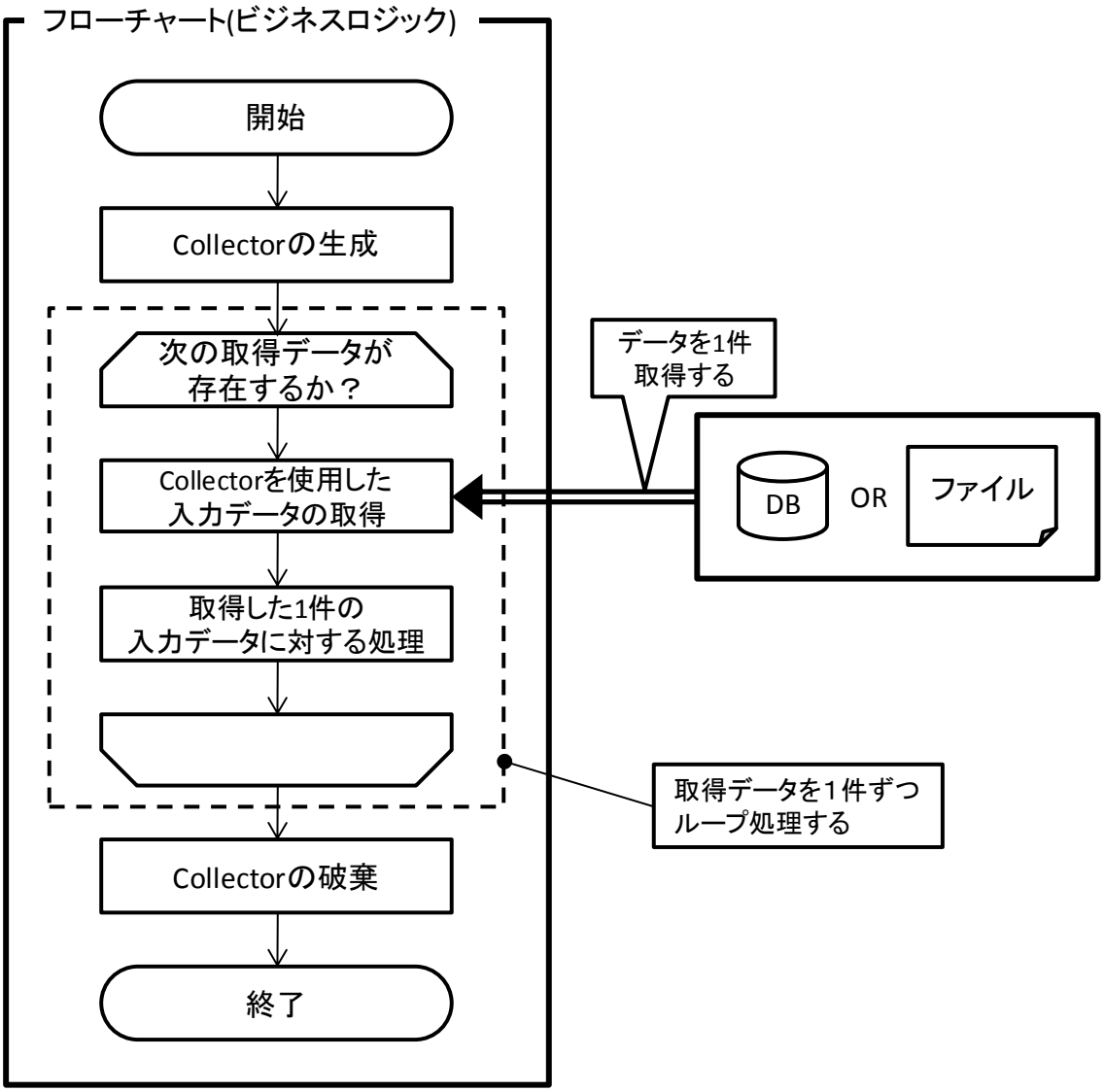
- 入力データ取得機能の機能説明書では、機能を以下のように細分化し、それぞれの機能説明書で解説するものとする。
  - 入力データ取得機能(本機能説明書)
  - コントロールブレイク機能
  - 入力チェック機能

  入力データ取得機能の説明書では、入力データ取得機能の基本的な使用方法である、入力データの取得部分についての解説を行う。

### ◆ 機能概要

- DB やファイルから入力データを取得する際に使用する機能である。
- DB からのデータ取得時に QueryRowHandleDAO を使用してデータを 1 件ずつ取得する。
- QueryDAO を使用してデータを一括で取得する場合と比べ、ヒープメモリの消費量を抑制する事ができる。
- ファイルからのデータ取得時には FileQueryDAO を使用し、ファイルを 1 行ずつ取得する。
- 入力データ取得機能を使用する事により、ビジネスロジックを構造化プログラミングで作成できる。

## ◆ 概念図




## ◆ 解説

- ビジネスロジックで、DB やファイルからデータを 1 件毎に取得する。
  - ビジネスロジックでは、初めに Collector インスタンスを生成し、以降はループ処理の中で取得したデータに対する処理を記述する。
  - データ取得部分を担当する Collector クラスは本機能が提供している。
  - DB から大量データを処理する場合であっても、ビジネスロジックではデータを 1 件ずつ扱うため、ヒープ領域の圧迫を抑制する事ができる。
    結果、取得するデータ量に因る性能への影響を極力抑える事ができる。

## ◆ コーディングポイント

【コーディングポイントの構成】

- ビジネスロジックの実装例
  - データベースからデータを取得する際のビジネスロジックの実装例
  - CSV ファイルからデータを取得する際のビジネスロジックの実装例
    (データ取得時の例外発生時に、処理を停止するパターン)
  - CSV ファイルからデータを取得する際のビジネスロジックの実装例
    (データ取得時の例外発生時に、例外を無視して処理を継続するパターン)
- Collector クラスのコンストラクタについて
  - コンストラクタで設定できる内容について
  - Collector のコンストラクター一覧
  - コンストラクタ引数一覧

- ビジネスロジックの実装例
  以下に入力データ取得機能を利用して、データを取得する際の実装例を掲載する
  - データベースからデータを取得するビジネスロジックの実装例
    (TERASOLUNA Batch Framework for Java ver 3.x の場合)

```
@Component
public class Sample01BLogic extends AbstractTransactionBLogic {

    @Autowired
    protected QueryRowHandleDAO queryRowHandleDAO;

    @Override
    public int doMain(BLogicParam param) {

        // Collector の生成
        Collector<Sample01Bean> collector = new DBCollector<Sample01Bean>(
                this.queryRowHandleDAO, "Sample.selectData01", null);

        try {
            Sample01Bean inputData = null;
            while (collector.hasNext()) {
                // データの取得
                inputData = collector.next();

                // ファイルの出力など、取得データに対する処理を記述する(実装は省略)
            }
        } catch (Exception e) {
            // 例外処理
        } finally {
            // Collector のクローズ
            CollectorUtility.closeQuietly(collector);
        }
        return 0;
    }
}
```

DB からのデータ取得時は QueryRowHandleDAO を使用する。

コンストラクタを使ってコレクタを生成する。
第一引数:QueryRowHandleDAO
第二引数:SQLID(文字列)
第三引数:SQL にバインドされるパラメータ

while 文を使用して次のデータが存在する限り取得データに対してループ処理を行う。

必ず処理の最後にコレクタをクローズする事

- CSV ファイルからデータを取得するビジネスロジックの実装例
  (データ取得時の例外発生時に、処理を停止するパターン)
  (TERASOLUNA Batch Framework for Java ver 3.x の場合)

```
@Component
public class Sample02BLogic extends AbstractTransactionBLogic {

    @Autowired
    @Qualifier(value = "csvFileQueryDAO")
    protected FileQueryDAO csvFileQueryDAO;

    @Override
    public int doMain(BLogicParam param) {

        // Collector の生成
        Collector<Sample02Bean> collector = new FileCollector <Sample02Bean>(
                this.csvFileQueryDAO, "inputFile/sinput_Sample02.csv", Sample02Bean.class);

        try {
            Sample02Bean inputData = null;
            while (collector.hasNext()) {
                // データの取得
                inputData = collector.next();

                // DB の更新など、取得データに対する処理を記述する(実装は省略)
            }
        } catch (Exception e) {
            // 例外処理
        } finally {
            // Collector のクローズ
            CollectorUtility.closeQuietly(collector);
        }
        return 0;
    }
}
```

CSV ファイルからのデータ取得時は CSVFileQueryDAO を使用する

コンストラクタを使ってコレクタを生成する。
第一引数: FileQueryDAO
第二引数:読み込むファイル名
第三引数:ファイル行オブジェクト

while 文を使用して次のデータが存在する限り取得データに対してループ処理を行う。

このパターンでは、データ取得時に発生した例外は、ここでキャッチされる。

必ず処理の最後にコレクタをクローズする事

ファイルの読み込みでは、読み込む際にフォーマットエラー等の例外が発生した場合に、例外を無視して処理を継続させる事が可能である。
次ページにデータ取得時の例外発生時に、例外を無視して処理を継続するパターンでの実装例を掲載する。

- CSV ファイルからデータを取得するビジネスロジックの実装例
  (データ取得時の例外発生時に、例外を無視して処理を継続するパターン)
  (TERASOLUNA Batch Framework for Java ver 3.x の場合)

```
@Component
public class Sample03BLogic extends AbstractTransactionBLogic {

    @Autowired
    @Qualifier(value = "csvFileQueryDAO")
    protected FileQueryDAO csvFileQueryDAO;

    @Override
    public int doMain(BLogicParam param) {

        // Collector の生成
        Collector<Sample03Bean> collector = new FileCollector< Sample03Bean >(
                this.csvFileQueryDAO, "inputFile/input_Sample03.csv", Sample03Bean.class);

        try {
            Sample03Bean inputData = null;
            while (collector.hasNext()) {
                try {
                    // データの取得
                    inputData = collector.next();
                } catch (FileException e) {
                    continue;
                }
                // DB の更新など、取得データに対する処理を記述する
            }
        } catch (Exception e) {
            // 例外処理
        } finally {
            // Collector のクローズ
            CollectorUtility.closeQuietly(collector);
        }
        return 0;
    }
}
```

データ取得部分を try-catch 文で囲む

ファイル読み込みに関する例外が発生した際は、例外はここでキャッチされ、以降の処理をスキップし、次のループ処理へと移行する。

必ず処理の最後にコレクタをクローズする事

  - このように実装することで、データ取得時に FileException(ファイルアクセスで発生する例外のラップクラス)が発生した場合に、その行を無視して処理を継続させる事が可能である。

- Collector クラスのコンストラクタについて
  DBCollector と FileCollector が用意するコンストラクタと、コンストラクタに使用される引数の一覧を掲載する。
  - コンストラクタで設定できる内容について
    実装例で使用した基本的なコンストラクタの他に、引数を与える事により、以下の項目を設定する事が可能である。
    - iBATIS の groupBy 属性使用の有無(DB のみ)(※１)
    - キューサイズ
    - 拡張例外ハンドラクラス(※２)

    ※1. iBATIS の groupBy 属性を使用する事によって、1:N 関係にあるテーブルの内容を、１つのクエリーで取得する事が出来る。
    詳細は iBATIS の機能説明書 P42 の「N+1 Selects を回避する」の項目を参照の事。(*http://ibatis.apache.org/docs/java/pdf/iBATIS-SqlMaps-2_ja.pdf*)

    ※2. 拡張例外ハンドラクラスに関しては、後述の拡張ポイントの項目を参照する事。
  - Collector のコンストラクター覧
    先ほどの番号と合わせて以下にコンストラクタを列挙し、概要を掲載する。
    引数についての詳細は、次ページのコンストラクタ引数一覧を参照する事。
    - DBCollector のコンストラクター覧

| コンストラクタ | 概要 |
|---|---|
| DBCollector<P>(QueryRowHandleDAO, String, Object) | 実装例で掲載した基本となるコンストラクタ<br>これら３つの引数は必須である。 |
| DBCollector<P>(QueryRowHandleDAO, String, Object, boolean) | 基本となるコンストラクタ及び、<br>1:N マッピング使用の有無を設定する。 |
| DBCollector<P>(QueryRowHandleDAO, String, Object, int) | 基本となるコンストラクタ及び、<br>キューサイズを設定する。 |
| DBCollector<P>(QueryRowHandleDAO, String, Object, CollectorExceptionHandler) | 基本となるコンストラクタ及び、<br>拡張例外ハンドラクラスを設定する。 |
| DBCollector<P>(QueryRowHandleDAO, String, Object, int, boolean, CollectorExceptionHandler) | 基本となるコンストラクタ及び、<br>1:N マッピング使用の有無、<br>キューサイズ、<br>拡張例外ハンドラクラスを設定する。 |

    - FileCollector のコンストラクター覧

| コンストラクタ | 概要 |
|---|---|
| FileCollector<P>(FileQueryDAO, String, Class<P>) | 実装例で掲載した基本となるコンストラクタ<br>これら３つの引数は必須である。 |
| FileCollector<P>(FileQueryDAO, String, Class<P>, CollectorExceptionHandler) | 基本となるコンストラクタ及び、<br>拡張例外ハンドラクラスを設定する。 |
| FileCollector<P>(FileQueryDAO, String, Class<P>, int, CollectorExceptionHandler) | 基本となるコンストラクタ及び、<br>キューサイズ、<br>拡張例外ハンドラクラスを設定する。 |

➢ コンストラクタ引数一覧

前ページで列挙したコンストラクタで使用される引数の一覧を掲載する。

✧ DBCollector のコンストラクタに渡される引数

| 引数 | 解説 | デフォルト値 | 省略 |
|---|---|---|---|
| QueryRowHandleDAO | DB にアクセスするための DAO | － | 不可 |
| String | SqlMap で定義した SQLID | － | 不可 |
| Object | SQL にバインドされる値を格納したオブジェクト、バインドする値が存在しない場合は省略せず、null を渡す事。 | － | 不可 |
| int | キューサイズ、0 以下の値は無視される。基本的に変更不要(※３) | 20 | 可 |
| CollectorExceptionHandler | 拡張例外ハンドラクラス | null | 可 |
| boolean | iBATIS の 1:N マッピング使用時は true を渡す。<br>true にする事により、メモリの肥大化を最小限に抑えることができる。 | false | 可 |

✧ FileCollector のコンストラクタに渡される引数

| 引数 | 解説 | デフォルト値 | 省略 |
|---|---|---|---|
| FileQueryDAO | ファイルにアクセスするための DAO | － | 不可 |
| String | 読み込むファイル名 | － | 不可 |
| Class<P> | ファイル行オブジェクトクラス | － | 不可 |
| int | キューサイズ、0 以下の値は無視される。基本的に変更不要(※３) | 20 | 可 |
| CollectorExceptionHandler | 拡張例外ハンドラクラス | null | 可 |

※3. キューサイズの変更方針

基本的にはデフォルトの 20 から変更する必要はない。

デフォルト値で以下のような問題が発生する場合のみ変更する事。

✧ キューオブジェクトのサイズが大きすぎてヒープ領域を圧迫するような場合。
デフォルトの 20 からキューサイズを減少させる。

✧ キューサイズが小さすぎて、性能が低下している場合は、デフォルトの 20 からキューサイズを増加させる。
（ただし、キューサイズと性能は比例するものではないので、増やせば性能が向上するというものではない。自環境で十分な性能を発揮できる値を探し、設定する事）

## ◆ 拡張ポイント

- 独自実装した拡張例外ハンドリングを行う方法(ファイルからのデータ取得のみ)
  拡張例外ハンドラクラスを独自実装する事により、ログの出力方法・ログレベルを変更や、処理の継続・中断といった挙動の制御が可能になる。
  - ➢ 拡張例外ハンドラクラスの実装例
    以下の仕様を満たす拡張例外ハンドラクラスの実装例を掲載する。
    【仕様】
    ① ログレベル ERROR で発生した例外情報をコンソールに残す。
    ② 例外発生後も処理は継続する。

```
public class CustomCollectorExceptionHandler implements CollectorExceptionHandler {

    private Log log = LogFactory.getLog(CustomCollectorExceptionHandler.class);

    @Override
    public CollectorExceptionHandlerStatus handleException(
            DataValueObject dataValueObject) {

        // ログ出力
        if(log.isErrorEnabled()){
            log.error("以下の例外が発生しました。" , dataValueObject.getThrowable());
        }

        // CollectorExceptionHandlerStatus の設定
        return CollectorExceptionHandlerStatus.SKIP;
    }

}
```

handleException メソッドの実装を行う。

CollectorExceptionHandler インタフェースの実装を行う。

このメソッドは引数にエラー情報が格納された DataValueObject を持っている。

DataValueObject が持つエラー情報を利用したログ出力

ステータス:CollectorExceptionHandlerStatus.SKIP を返却することにより以降の処理を継続する。(ステータスについては下表を参照)

  - ➢ 拡張例外ハンドラクラスが返却するステータスについて
    返却するステータスによって、例外発生後の Collector の挙動が制御される。
    - ✧ ステータス：CollectorExceptionHandlerStatus 一覧表

| CollectorExceptionHandlerStatus | 挙動 |
|---|---|
| **SKIP** | 発生した例外はスローせず、<br>以降の処理を**継続**する。<br>(上の実装例にて使用) |
| **END** | 発生した例外をスローせずに、<br>以降の処理を**停止**する。 |
| **THROW** | 発生した例外をそのままスローする |

  - ➢ 拡張例外ハンドラクラスを使用する場合のビジネスロジック実装例
    ビジネスロジック中で Collector のインスタンスを生成するタイミングでコンストラクタを利用して、拡張例外ハンドラクラスを渡しておく必要がある。

ここでは FileCollector インスタンスの生成時のコーディングのみ記載し、ビジネスロジック全体像については掲載しない。

✧ ビジネスロジック実装例(FileCollector 生成部分のみ抜粋)

```
(略)

// FileCollector の生成
Collector<Sample01Bean> collector = new FileCollector<Sample01Bean>(
      this.csvFileQueryDAO, "inputFile/input_Sample1.csv",
      SampleFileLineObject.class, new CustomCollectorExceptionHandler());

(略)
```

コレクタ生成時に、先ほどの実装例で作成した拡張例外ハンドラクラスを渡してやる。

以上のように FileCollector のインスタンスを生成する事により、Collector.getNext()によるデータ取得時に例外が発生した場合、自動的に拡張例外ハンドラクラスの handleException メソッドが呼び出され、独自実装された例外ハンドリング処理を行う。

➢ 拡張例外ハンドリングについての補足事項

✧ DBCollector での本機能の使用について。

DB からのデータの取得時にも本機能を使用する事は可能だが、ファイルからのデータの取得の場合と異なり、拡張例外ハンドラクラスにて、ステータス[CollectorExceptionHandlerStatuscollector.SKIP]を返却しても、例外発生後に Collector の処理を継続させる事は不可能である。

● Collector クラスを独自実装する方法

➢ 本機能が提供する AbstractCollector クラスの拡張クラスを作成する事によって Collector クラスを独自実装する事が出来る。

➢ 実装方法については、本機能が提供する FileCollector の実装を参考にする事。

# ■ リファレンス

## ◆ 構成クラス

| | クラス名 | 概要 |
|---|---|---|
| 1 | jp.terasoluna.fw.collector.Collector | Iterator インタフェースなどを拡張した Collector インタフェースクラス<br>現在の要素と前後の要素にアクセスするためのメソッドを宣言している。 |
| 2 | jp.terasoluna.fw.collector.AbstractCollector | Collecotr インタフェースを実装した抽象クラス<br>キューを保持するフィールドなどを持たせ、入力データ取得機能のメイン部分の処理をコーディングしている。 |
| 3 | jp.terasoluna.fw.collector.db.DBCollector | DB 用の AbstractCollector 拡張クラス<br>DB からデータを取得する際に使用する。 |
| 4 | jp.terasoluna.fw.collector.file.FileCollector | ファイル用の AbstractCollector 拡張クラス<br>ファイルからデータを取得する際に使用する。 |
| 5 | jp.terasoluna.fw.collector.db.QueueingDataRowHandler | DataRowHandler インタフェースの拡張インタフェースクラス<br>DB からデータを 1 件ずつ処理するためのいくつかのメソッドの宣言を行っている。 |
| 6 | jp.terasoluna.fw.collector.db.QueueingDataRowHandlerImpl | QueueingDataRowHandler 実装クラス<br>取得したデータをキューに詰める部分などを実装したクラス。 |
| 7 | jp.terasoluna.fw.collector.db.Queueing1NRelationDataRowHandlerImpl | QueueingDataRowHandler 実装クラス<br>iBATIS の 1:N マッピングを利用する場合に使用する QueueingDataRowHandlerImpl の拡張クラスでもある。 |
| 8 | jp.terasoluna.fw.collector.vo.DataValueObject | キューに詰められる ValueObject、取得したデータや取得時に発生した例外などが格納される。 |
| 9 | jp.terasoluna.fw.collector.CollectorThreadFactory | データの取得を行う別スレッドを生成するためのスレッドファクトリクラス |

## ◆ 関連機能

- 『BB-01 データベースアクセス機能』
- 『BC-01 ファイルアクセス機能』

## ◆ 使用例

- **機能網羅サンプル**(terasoluna-batch-functionsample)
- **チュートリアル**(terasoluna-batch-tutorial)

## ◆ 備考

- 大量データを検索する際の注意事項
  iBATIS マッピング定義ファイルの<statement> 要素、<select> 要素、<procedure> 要素にて大量データを返すようなクエリを記述する場合には、fetchSize 属性に適切な値を設定しておくこと。fetchSize 属性には JDBC ドライバとデータベース間の通信において、一度の通信で取得するデータの件数を設定する。fetchSize 属性を省略した場合は各 JDBC ドライバのデフォルト値が利用される。

  ➢ SQL 定義ファイルの記述例

```
(略)

<select id=”sq0001” resultClass=”jp.terasoluna.xxx.bean.SampleXXXBean” fetchSize=”50”>
      SELECT ID,FAMILYNAME,FIRSTNAME,AGE FROM USER
</select>

(略)
```

大量データ取得時には、fetchSize を明示的に設定しておく。

※例えば PostgreSQL JDBC ドライバ(postgresql-8.3-604.jdbc3.jar にて確認) のデフォルトは、一度の通信で検索対象のデータが全件取得される。数十件程度の件数であれば問題にならないが、数万件以上の大量データを検索する場合にはヒープメモリを圧迫してしまう可能性がある。

# AL-042　コントロールブレイク機能

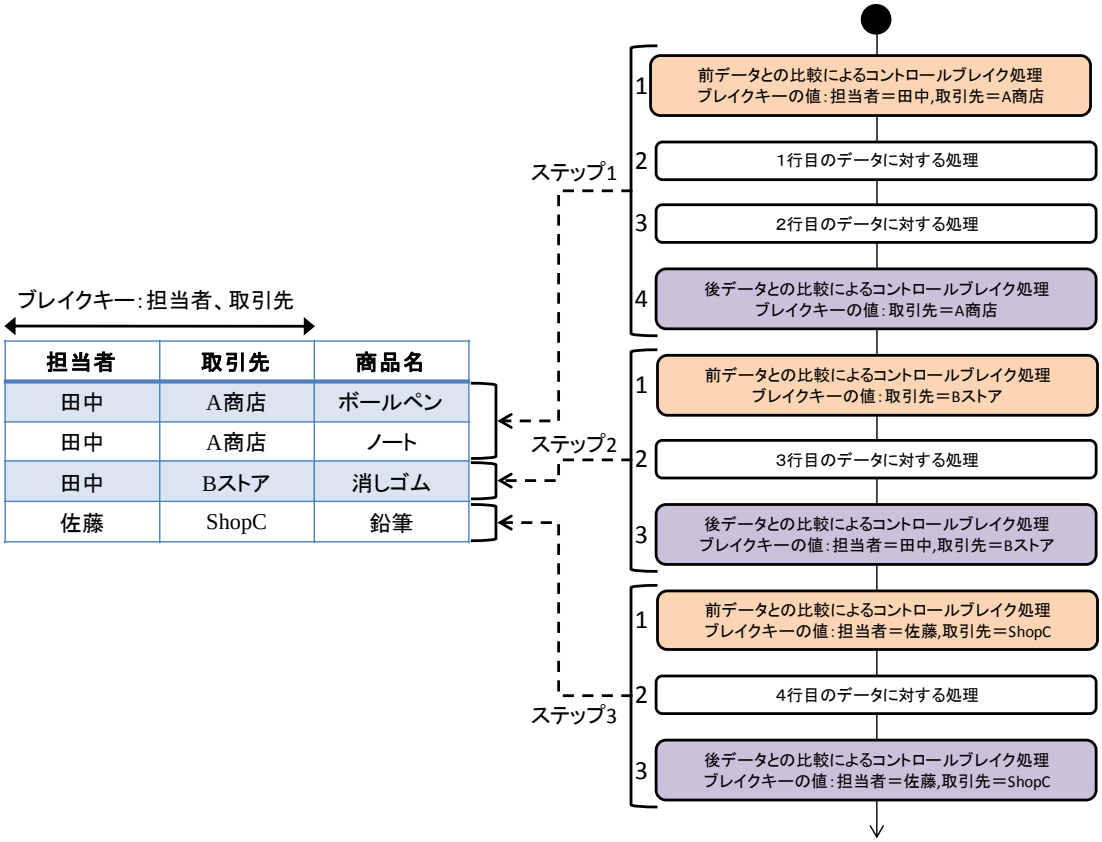
## ■ 概要

### ◆ 機能概要

- コントロールブレイク処理とは、ある項目をキーとして、キーが変わるまで集計したり見出しを追加したりする処理である。キーブレイク処理とも呼ばれる。
- 本機能では、コントロールブレイク処理を行うためのユーティリティを提供する。
- コントロールブレイクの判断には「前データとの比較」「後データとの比較」の二種類を用意している。
  - 取得したデータの処理前にコントロールブレイク処理を行いたい場合は「前データとの比較」を使用する。(例：見出しの作成など)
  - 取得したデータの処理後にコントロールブレイク処理を行いたい場合は「後データとの比較」を使用する。(例：データの集計など)

※ 本機能を使用する際には、「AL-041 入力データ取得機能」の使用が前提条件となる。

## ◆ 概念図




## ◆ 解説

| ステップ | | 処理内容 |
|---|---|---|
| 1 | 1 | 前データとの比較によるコントロールブレイク処理が実行される。 |
| | 2 | 1 行目に対する処理が実行される。 |
| | 3 | 2 行目に対する処理が実行される。 |
| | 4 | 後データとの比較によるコントロールブレイク処理が実行される |
| 2 | 1 | 前データとの比較によるコントロールブレイク処理が実行される。 |
| | 2 | 3 行目に対する処理が実行される。 |
| | 3 | 後データとの比較によるコントロールブレイク処理が実行される |
| 3 | 1 | 前データとの比較によるコントロールブレイク処理が実行される。 |
| | 2 | 4 行目に対する処理が実行される。 |
| | 3 | 後データとの比較によるコントロールブレイク処理が実行される |

- 設定したブレイクキーの値が切り替わったタイミングでコントロールブレイク処理が実行される。
- ビジネスロジック中でブレイクキーを取得する事により、ビジネスロジック中で切り替わった値を取得する事が可能である。

## ◆ コーディングポイント

【コーディングポイントの構成】

- ファイル行オブジェクトクラスの実装例
- コントロールブレイク処理の実装例(単一のコントロールブレイクキー)
  - ビジネスロジックの処理イメージ
  - ビジネスロジックの実装例
- コントロールブレイク処理の実装例(複数のコントロールブレイクキー)
  - ビジネスロジックの処理イメージ
  - ビジネスロジックの実装例
- コントロールブレイク機能を使用する場合の注意点
- コントロールブレイク機能が持つメソッドの一覧、及び解説
  - ControlBreakChecker クラスのメソッド一覧
  - ControlBreakChecker クラスのメソッドで使用する引数一覧

- ファイル行オブジェクトクラスの実装例
  以降のビジネスロジックの実装例で使用するファイル行オブジェクトの実装例を掲載する。
  ファイル行オブジェクトクラスは以下の二つの役割を持っている。
  - DB から取得した１行分のデータをマッピングさせるオブジェクト
  - １行分のデータをファイルに出力するためのファイル行オブジェクト

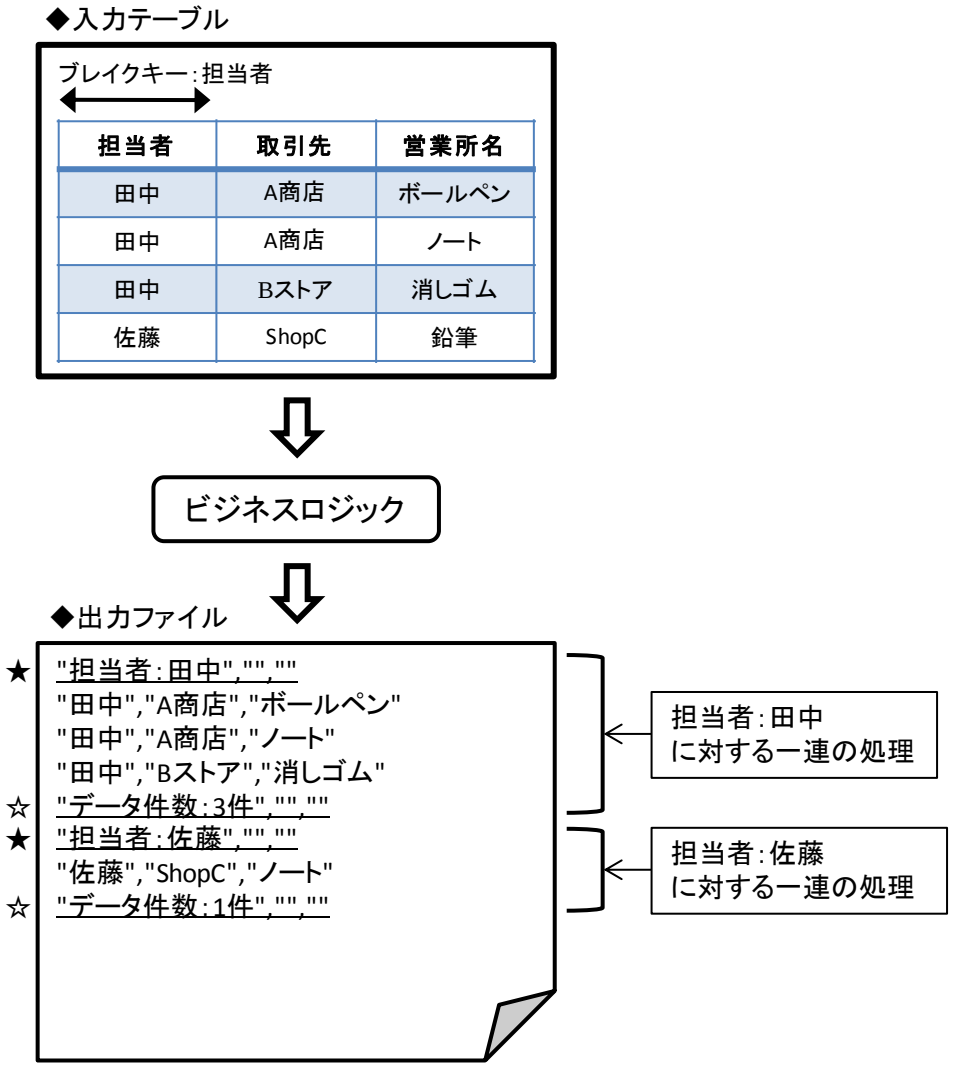
```
@FileFormat(encloseChar = "", overWriteFlg = true)
public class CBData {

    @OutputFileColumn(columnIndex = 0)
    private String tantousya = null;

    @OutputFileColumn(columnIndex = 1)
    private String shopName = null;

    @OutputFileColumn(columnIndex = 2)
    private String itemName = null;

    (setter/getter は特筆する点が無いので省略する)
```

囲み文字、上書きフラグの設定

フィールドとカラム番号の紐付け

その他、ファイル出力に関するアノテーションについての詳細は「BC-01 ファイルアクセス機能」の機能説明書を参照の事

- コントロールブレイク処理の実装例(単一のコントロールブレイクキー)
  概念図の表に対して、コントロールブレイクキーを「担当者」のみに絞った場合のビジネスロジックの実装例を以下に掲載する
  - ビジネスロジックの処理イメージ




図中でアンダーラインを引いている行は、コントロールブレイク処理によって出力された行である。

※ 実装イメージ中の記号の意味

★…前データとの比較によるコントロールブレイク処理による出力

☆…後データとの比較によるコントロールブレイク処理による出力

次ページで上記イメージ図のビジネスロジックの実装例を解説と一緒に掲載する。

➢ ビジネスロジックの実装例
(TERASOLUNA Batch Framework for Java ver 3.x の場合)

```
@Component
public class B101BLogic extends AbstractTransactionBLogic {

    @Autowired
    protected QueryRowHandleDAO queryRowHandleDAO;

    @Autowired
    @Qualifier(value = "csvFileUpdateDAO")
    protected FileUpdateDAO csvFileUpdateDAO;

    @Override
    public int doMain(BLogicParam param) {
        // コレクタの生成
        Collector<CBData> collector = new DBCollector<CBData>(
                this.queryRowHandleDAO, "Common.selectData01", null);

        // ファイル出力用行ライタの取得
        FileLineWriter<CBData> fileLineWriter = csvFileUpdateDAO.execute(
                "outputFile/SampleOutput01.csv", CBData.class);

        // データ件数カウント用
        private int dataCount = 0;
        // コントロールブレイクキーの設定
        String cbKey = "tantousya";

        try {
            while (collector.hasNext()) {
                // データの取得
                CBData inputData = collector.next();
                // コントロールブレイク判断（前データとの比較）
                if (ControlBreakChecker.isPreBreak(collector, cbKey)) {
                    // コントロールブレイクキー取得
                    Map<String, Object> keyMap = ControlBreakChecker
                            .getPreBreakKey(collector, cbKey);
                    // コントロールブレイク処理
                    fileLineWriter.printDataLine(new CBData("担当者:"
                            + keyMap.get(cbKey).toString(), "",    ""));
                }
```

(次ページに続く)

String 型でコントロール
ブレイクキーの設定を行う。

前データとの比較の際は
isPreBreak メソッドを使用する。

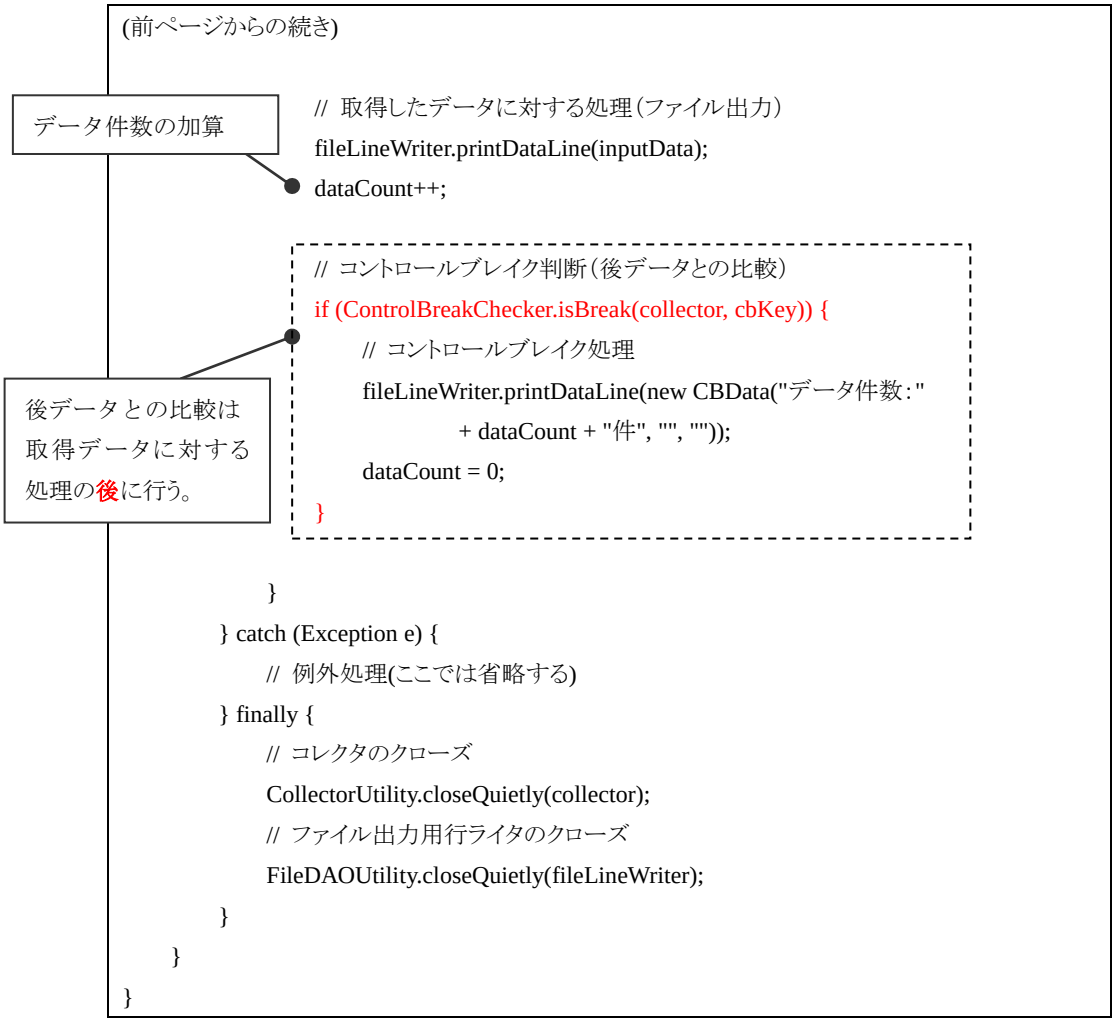
前データとの比較は
取得データに対する
処理の**手前**で行う。

getPreBreakKey メソッドで切り替わった値を Map 型で取得できる。
具体例(4行目のデータ処理前)

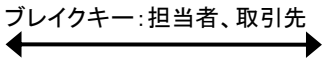
keyMap

| key | value |
|---|---|
| tantousya | 佐藤 |

今回は取得した Map から
担当者名を取得し、
ヘッダに出力している。



- コントロールブレイク処理の実装例(複数のコントロールブレイクキー)
  - コントロールブレイクキーを複数設定した場合は、上位のキーは下位のキーを包含する関係となる。
  - すなわち、上位のキーが切り替わった際、例え下位のキーが切り替わっていなくとも、getPreBreakKey(もしくは getBreakKey)メソッドを呼び出すと、上位のコントロールブレイクキーの値と同時に、下位のコントロールブレイクキーの値を取得できる。

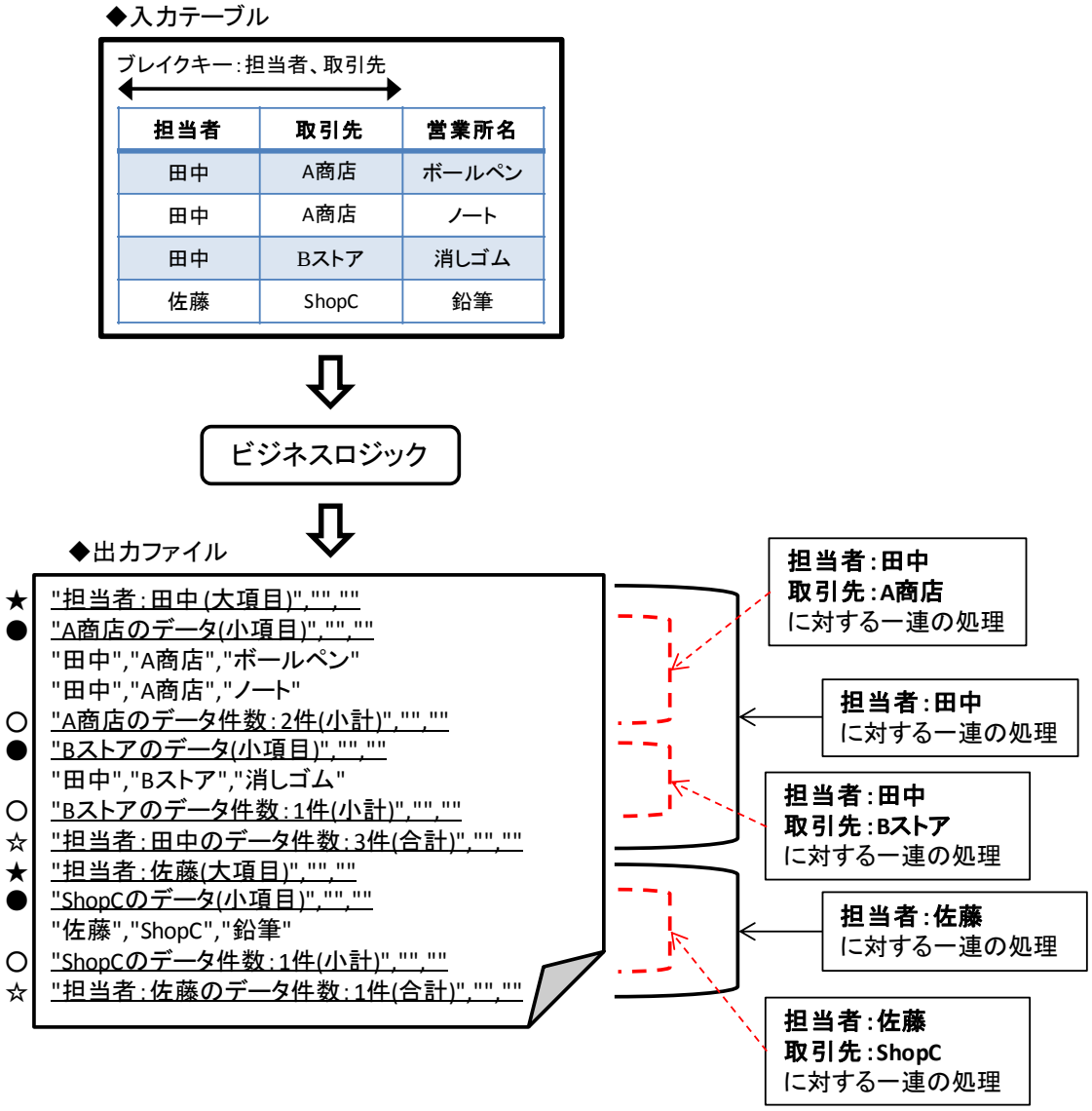



| 担当者 | 取引先 | 営業所名 |
|---|---|---|
| 田中 | A商店 | ボールペン |
| 佐藤 | Bストア | ノート |
| 佐藤 | ShopC | 消しゴム |
| 鈴木 | ShopC | 鉛筆 |

  - 上記の表に対して具体的な例を挙げる。
    - 上記の表では 3 行目と 4 行目の間で上位キーである担当者の値が「佐藤」から「鈴木」へと切り替わるが、下位キーである取引先の値は、3 行目も 4 行目も「ShopC」のままである。
    - しかしながら、上位キーは下位キーと包含関係にあるため、担当者の切り替わる時に getPreBreakKey メソッド、もしくは getBreakKey メソッドを呼び出した場合は「担当者」の値だけでなく、実際には値の変更の無い「取引先」の値も取得できる。
    - その結果、担当者・取引先の二つのコントロールブレイクキーを取得し、取得したそれぞれのキーに対してコントロールブレイク処理を実施する事が可能である。

  - ブレイクキーを「担当者」「取引先」の二つを設定した場合のビジネスロジックの処理イメージ・実装例を次ページから掲載する。

➢ ビジネスロジックの処理イメージ(複数のコントロールブレイクキー)




※ 実装イメージ中の記号の意味

★…担当者、取引先が切り替わった時に動作する、「**前データ**との比較」によるコントロールブレイク処理による出力行

●…取引先が切り替わった時に動作する、「**前データ**との比較」によるコントロールブレイク処理による出力行

☆…担当者、取引先が切り替わった時に動作する、「**後データ**との比較」によるコントロールブレイク処理による出力行

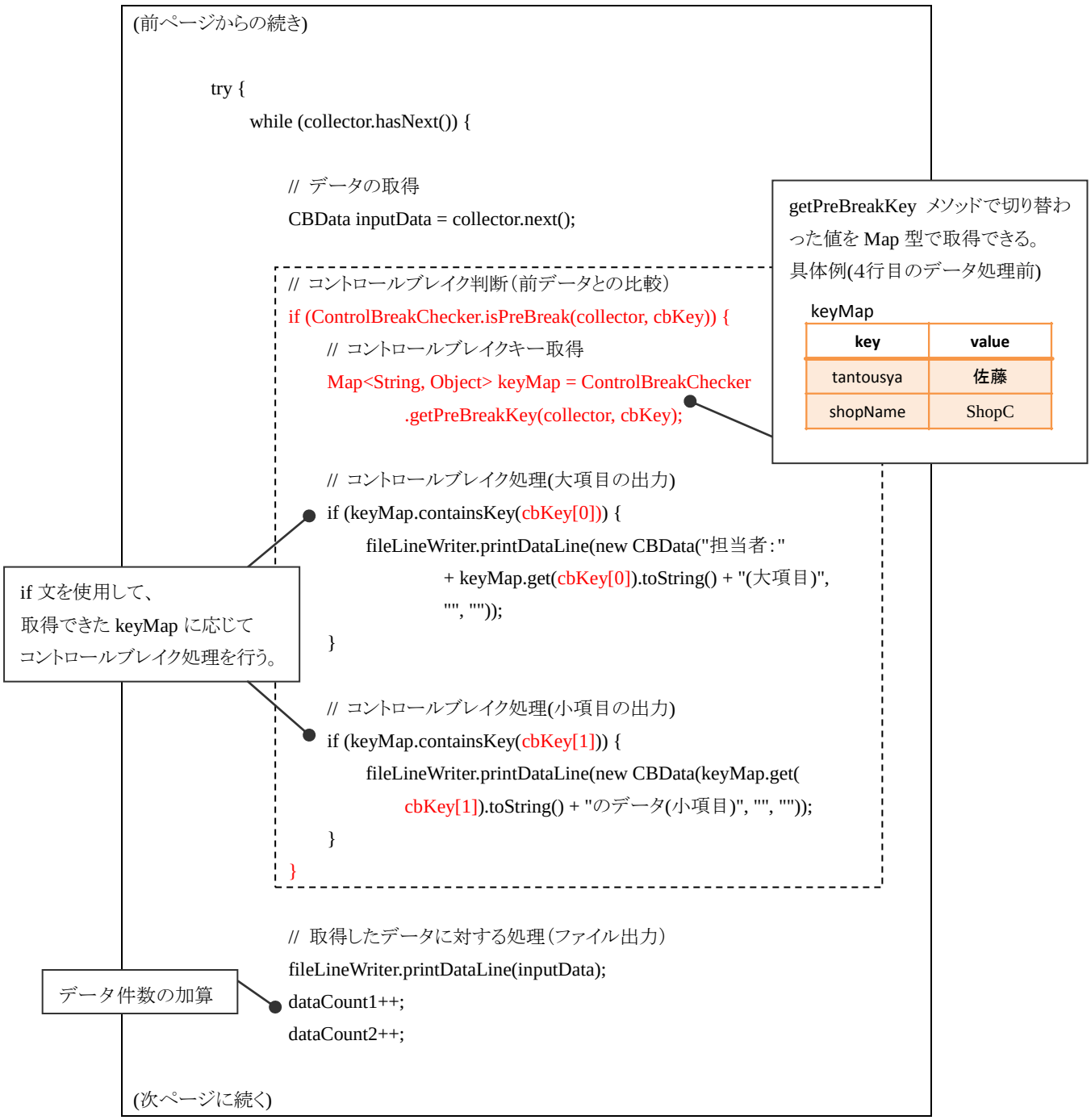
○…取引先が切り替わった時に動作する、「**後データ**との比較」によるコントロールブレイク処理による出力行

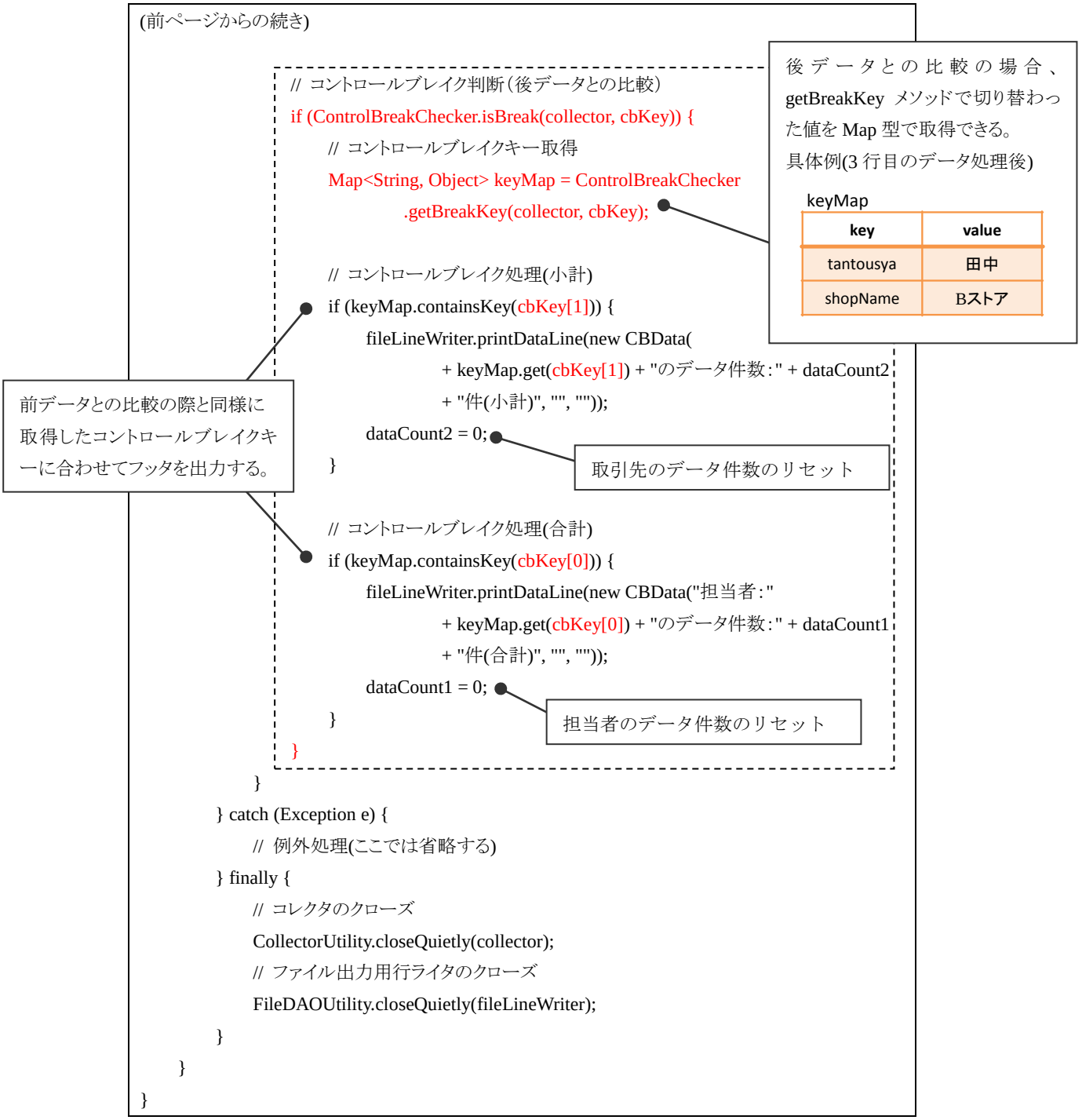
➢ コントロールブレイク処理の実装例(複数のコントロールブレイクキー)
(TERASOLUNA Batch Framework for Java ver 3.x の場合)

```
@Component
public class B102BLogic extends AbstractTransactionBLogic {

    @Autowired
    protected QueryRowHandleDAO queryRowHandleDAO;

    @Autowired
    @Qualifier(value = "csvFileUpdateDAO")
    protected FileUpdateDAO csvFileUpdateDAO;

    @Override
    public int doMain(BLogicParam param) {

        // コレクタの生成
        Collector<CBData> collector = new DBCollector<CBData>(
                this.queryRowHandleDAO, "Common.selectData02", null);

        // ファイル出力用行ライタの取得
        FileLineWriter<CBData> fileLineWriter = csvFileUpdateDAO.execute(
                "outputFile/SampleOutput02.csv", CBData.class);

        // 担当者データ件数カウント用(合計用)
        private int dataCount1 = 0;

        // 取引先データ件数カウント用(小計用)
        private int dataCount2 = 0;

        // コントロールブレイクキーの設定
        String[] cbKey = new String[] { "tantousya", "shopName" };
```

(次ページに続く)

複数のコントロールブレイクキーを設定する場合は、
String 型の配列で生成する





- コントロールブレイク機能が持つメソッドの一覧、及び解説
  - ControlBreakChecker クラスのメソッド一覧

| メソッド名 | 引数 | 戻り値 | 解説 |
|---|---|---|---|
| isPreBreak | (Collector<?>, String...) | boolean | 前データとの比較により、コントロールブレイクを判断するメソッド |
| isBreak | (Collector<?>, String...) | boolean | 後データとの比較により、コントロールブレイクを判断するメソッド |
| getPreBreakKey | (Collector<?>, String...) | Map<String, Object> | 前データとの比較の際に、コントロールブレイクキーを取得するメソッド |
| getBreakKey | (Collector<?>, String...) | Map<String, Object> | 後データとの比較の際に、コントロールブレイクキーを取得するメソッド |

  - ControlBreakChecker クラスのメソッドで使用する引数一覧

| 引数 | 解説 | 省略 |
|---|---|---|
| Collector<?> | DB コレクタや、ファイルコレクタなどのコレクタ実装クラス。 | 不可 |
| String… | コントロールブレイクキー、複数のブレイクキーを設定する際は、String の配列型で渡す。 | 不可 |

### ◆ 拡張ポイント

なし

## ■ リファレンス

### ◆ 構成クラス

| | クラス名 | 概要 |
|---|---|---|
| 1 | jp.terasoluna.fw.collector.util.ControlBreakChecker | コントロールブレイク判定クラス<br>前データとの比較と後データとの比較の 2 種類の方法により、コントロールブレイクを判断する。 |

### ◆ 関連機能

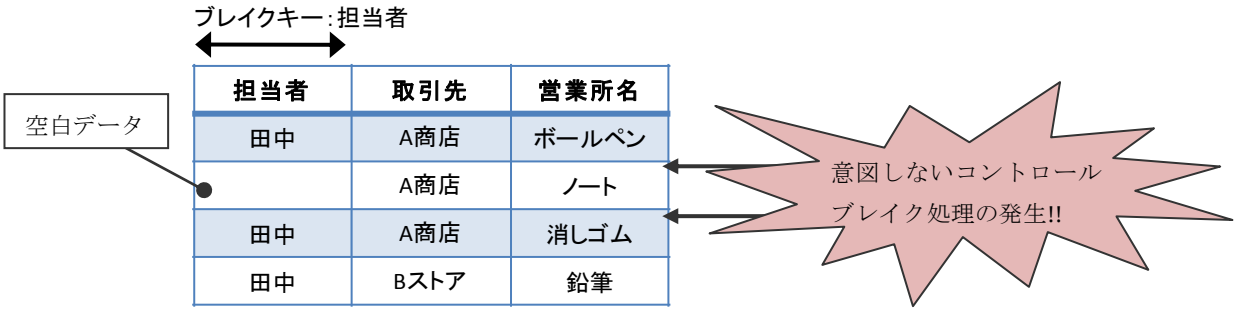
- 『AL-041 入力データ取得機能』
- 『AL-043 入力チェック機能』

## ◆ 使用例

- **機能網羅サンプル**(terasoluna-batch-functionsample)

## ◆ 備考

- コントロールブレイク機能を使用する場合の注意点
  コントロールブレイクキーに設定するカラムに不正なデータや空白が存在した場合、意図しないタイミングでコントロールブレイク処理が実施されてしまう。




  - 本機能を使用する際に、「AL-043 入力チェック機能」を併用して使用する事により、このような意図しないタイミングでのコントロールブレイク処理を避ける事が出来る。
  - 入力チェック機能についての説明は、「AL-043 入力チェック機能」の機能説明書を参照する事。

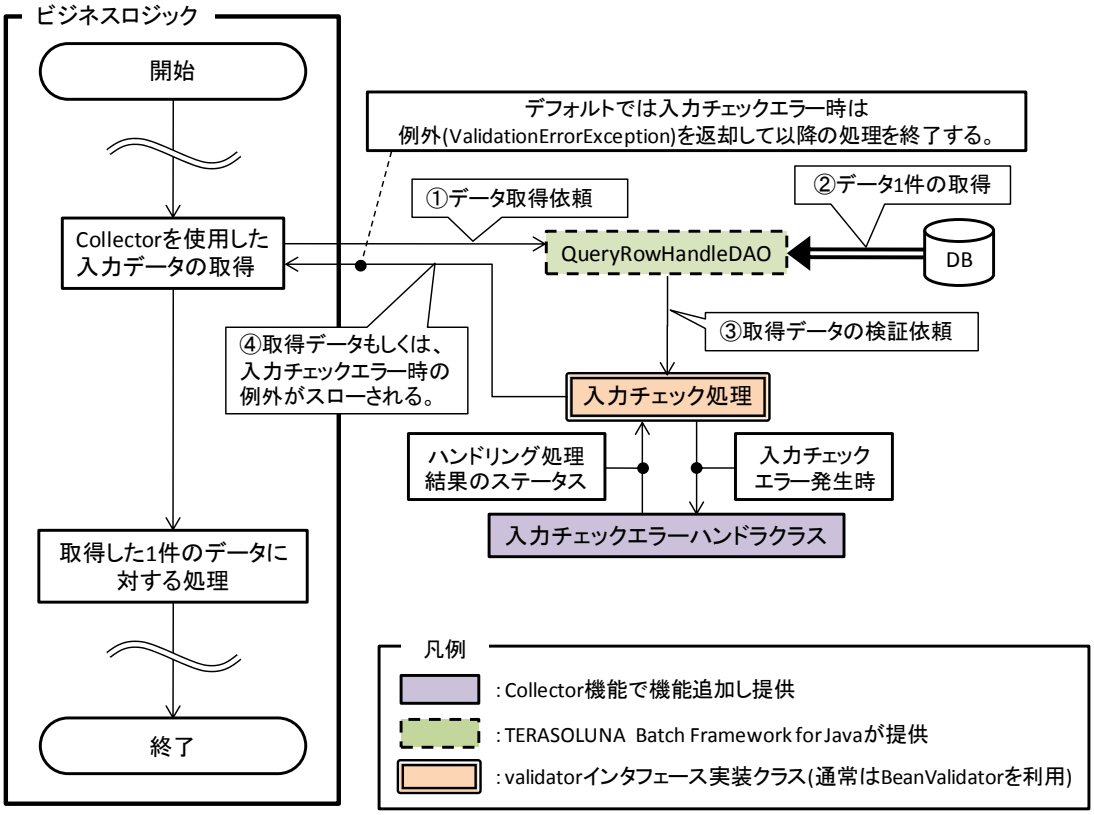
# AL-043　入力チェック機能

## ■ 概要

### ◆ 機能概要

- 「AL-041 入力データ取得機能」を使用した際に、DB やファイルから取得したデータ 1 件毎に入力チェックを行う機能を提供する。
- 入力チェックは、DB やファイルからデータを取得するタイミングで行われる。
- 本機能を使用する事で、コントロールブレイク判断に空レコードが渡される事の回避が可能となる。
- 本機能では、TERASOLUNA Server Framework for Java (Rich 版)の機能である「RF-02 入力チェック機能」と同様の入力チェックを実施する事が出来る。

## ◆ 概念図




## ◆ 解説

① Collector はデータの取得を QueryRowHandleDAO に依頼する。
② QueryRowHandleDAO は DB からデータを 1 件取得する。
③ QueryRowHandleDAO は取得したデータを返却する前に、validator インタフェース実装クラスに入力チェック処理を依頼する。
④ validator インタフェース実装クラスは入力チェックの結果に応じて、処理を振り分ける。
- 正常系の場合…取得データをビジネスロジックに返却する。
- 異常系の場合…入力チェックエラーハンドラクラスによって入力チェックエラー時の例外「ValidationErrorException」がビジネスロジックに返却される。

- この時、独自に作成した拡張入力チェックエラーハンドラクラスを使用する事によって、例外「ValidationErrorException」をスローする事なく以降の処理を継続させることも可能である。
- 拡張入力チェックエラーハンドラクラスを作成する場合は、拡張ポイントの項目を参照する事。

## ◆ コーディングポイント

【コーディングポイントの構成】

- 入力チェックを行う場合のビジネスロジックの実装例
  - ビジネスロジックの実装例(DB からのデータ取得)
  - ビジネスロジックの実装例(ファイルからのデータ取得)
- validator.xml(入力チェックルール)の設定例
  - validation.xml の設定例
  - form 名にクラス名を使用する方法
  - useFullyQualifiedClassName を false に設定した場合の form 名の記入例
- 本機能が提供する、入力チェックエラーハンドラクラスについて
- 入力チェック対応 Collector クラスのコンストラクタについて
  - コンストラクタで設定できる内容について
  - 入力チェック対応 Collector クラスのコンストラクタ一覧
  - コンストラクタ引数一覧

- 入力チェックを行う場合のビジネスロジックの実装例
  以下に DB からデータを取得する際に入力チェックを行う際の実装例を掲載する。
  使用する Collector クラスが、入力チェックを行わない場合と異なる点に注意する。
  - ビジネスロジックの実装例(DB からのデータ取得)
    (TERASOLUNA Batch Framework for Java ver 3.x の場合)

```
@Component
public class Sample01BLogic extends AbstractTransactionBLogic {

    @Autowired
    protected QueryRowHandleDAO queryRowHandleDAO;

    @Autowired
    protected Validator validator;

    @Override
    public int doMain(BLogicParam param) {

        // Collector の生成
        Collector<Sample01Bean> collector = new DBValidateCollector<Sample01Bean>(
                this.queryRowHandleDAO, "Sample.selectData01", null, validator);
        try {
            Sample01Bean inputData = null;
            while (collector.hasNext()) {
                // データの取得
                inputData = collector.next();

                // 取得データに対する処理(ここでは省略する)
            }
        } catch (Exception e) {
            // 例外処理
        } finally {
            // Collector のクローズ
            CollectorUtility.closeQuietly(collector);
        }
        return 0;
    }
}
```

BeanValidator の DI を行う

DBValidateCollector を生成する。
DBCollector と異なり、第四引数に入力チェックを行う
BeanValidator クラスを渡している点に注意する事。

このタイミングで入力チェックが行われる。
入力チェックエラー発生時には、ハンドラクラス
ExceptionValidationErrorHandler によって、例外
「ValidationErrorException」をスローする。

必ず処理の最後にコレクタをクローズする事

- ビジネスロジックの実装例(ファイルからのデータ取得)
  (TERASOLUNA Batch Framework for Java ver 3.x の場合)

```
@Component
public class Sample02BLogic extends AbstractTransactionBLogic {

    @Autowired
    @Qualifier(value = "csvFileQueryDAO")
    protected FileQueryDAO csvFileQueryDAO;

    @Autowired
    protected Validator validator;

    @Override
    public int doMain(BLogicParam param) {
        // Collector の生成
        Collector<Sample02Bean> collector = new FileValidateCollector<Sample02Bean>(
                this.csvFileQueryDAO, "inputFile/SampleFile.csv", Sample02Bean.class, validator);
        try {
            Sample02Bean inputData = null;
            while (collector.hasNext()) {
                // データの取得
                inputData = collector.next();

                // DB の更新など、取得データに対する処理を記述する(実装は省略)
            }
        } catch (Exception e) {
            // 例外処理
        } finally {
            // Collector のクローズ
            CollectorUtility.closeQuietly(collector);
        }
        return 0;
    }
}
```

BeanValidator の DI を行う

FileValidateCollector を生成する。
FileCollector と異なり、第四引数に入力チェックを行う
BeanValidator クラスを渡している点に注意する事。

このタイミングで入力チェックが行われる。
入力チェックエラー発生時には、ハンドラクラス
ExceptionValidationErrorHandler によって、例外
「ValidationErrorException」をスローする。

必ず処理の最後に Collector をクローズする事

- validation.xml（入力チェックルール）の設定例
  先ほど実装例を掲載した「入力チェックを行う場合のビジネスロジックの実装例(DB)」に合わせた、入力チェックルールの設定例を以下に掲載する。
  - validation.xml の設定例

```
<form-validation>
  <formset>
    <form name="jp.terasoluna.batch.sample.b101.blogic.Sample01Bean">
      <field property="name" depends="required" />
    </form>
  </formset>
</form-validation>
```

Bean のフルパスが form 名として扱われる。
後述の方法により、クラス名でも設定できる。

この例ではキューに詰める Sample01Bean が持つ
フィールド「name」が入力必須である事を定義している。

  - form 名にクラス名を使用する方法
    beanValidator の Bean 定義の際に、プロパティ useFullyQualifiedClassName を false に設定する事により、form 名をフルパスで扱う事が可能になる。
    以下に実装例を掲載する。
    - Bean 定義ファイル設定例(一部抜粋)

```
<!-- デフォルト入力チェッククラス クラス名をform名として扱う -->
<bean id="beanValidator"
   class="org.springmodules.validation.commons.DefaultBeanValidator">
   <property name="useFullyQualifiedClassName" value="false"/>
   <property name="validatorFactory" ref="validatorFactory"/>
</bean>
```

この部分を false に設定する

  - useFullyQualifiedClassName を false に設定した場合の form 名の記入例
    入力チェックルールの form 名にクラス名を使用する場合は、クラスの頭文字を小文字にする点に注意する。
    - validation.xml（入力チェックルール）の設定例(DB)

```
<form-validation>
  <formset>
    <form name="sample01Bean">
      <field property="name" depends="required" />
    </form>
  </formset>
</form-validation>
```

Sample01Bean でなく、sample01Bean と
なっている事に注意する事。

    validation.xml での入力チェックルールの定義方法については、この機能説明書では説明しない。
    以下の機能説明書を参考にする事。
    - TERASOLUNA Server Framework for Java (Rich 版)
      RF-02 入力チェック機能

- 本機能が提供する、入力チェックエラーハンドラクラスについて

| 入力チェックエラーハンドラクラス | 仕様 |
|---|---|
| ExceptionValidationErrorHandler | デフォルトで使用される入力チェックエラーハンドラクラス。入力チェックエラーが発生した時点で例外をスローし、以降の処理はすべて停止する。 |

- 入力チェック対応 Collecotr クラスのコンストラクタについて
  DBValidateCollector と FileValidateCollector が用意するコンストラクタと、コンストラクタに使用される引数の一覧を掲載する。
  - コンストラクタで設定できる内容について
    実装例で使用した基本的なコンストラクタの他に、引数を与える事により、以下の項目を設定する事が可能である。
    ① iBATIS の groupBy 属性使用の有無(DB のみ) (※１)
    ② キューサイズ
    ③ 拡張例外ハンドラクラス(※２)
    ④ 使用する入力チェックエラーハンドラクラス(※３)

    ※1. iBATIS の groupBy 属性を使用する事によって、1:N 関係にあるテーブルの内容を、１つのクエリーで取得する事が出来る。
    詳細は iBATIS の機能説明書 P42 の「N+1 Selects を回避する」の項目を参照の事。(*http://ibatis.apache.org/docs/java/pdf/iBATIS-SqlMaps-2_ja.pdf*)
    ※2. 拡張例外ハンドラクラスに関しては、「01 入力データ取得機能」の機能説明書の拡張ポイントの項目を参照する事。
    ※3. デフォルトでは先に紹介した「ExceptionValidationErrorHandler」が使用される、独自にハンドラクラスを作成する事も可能。
    ハンドラクラスを独自実装する場合は後述の拡張ポイントの項目を参照の事。

- 入力チェック対応 Collector クラスのコンストラクタ一覧
  先ほどの番号と合わせて以下にコンストラクタを列挙し、概要を掲載する。
  引数についての詳細は、次ページのコンストラクタ引数一覧を参照する事。
  - DBValidateCollector のコンストラクタ一覧

| コンストラクタ | 概要 |
|---|---|
| DBValidateCollector<P>(QueryRowHandleDAO, String, Object, Validator) | 実装例で掲載した基本となるコンストラクタ<br>これら４つの引数は必須である。 |
| DBValidateCollector<P>(QueryRowHandleDAO, String, Object, Validator, ValidationErrorHandler) | 基本となるコンストラクタ及び、<br>使用する入力チェックエラー<br>ハンドラクラスを設定する。 |
| DBValidateCollector<P>(QueryRowHandleDAO, String, Object, boolean, Validator) | 基本となるコンストラクタ及び、<br>1:N マッピング使用の有無を設定する。 |
| DBValidateCollector<P>(QueryRowHandleDAO, String, Object, boolean, Validator, ValidationErrorHandler) | 基本となるコンストラクタ及び、<br>1:N マッピング使用の有無、<br>使用する入力チェックエラーハンドラクラスを設定する。 |
| DBValidateCollector<P>(QueryRowHandleDAO, String, Object, int, Validator) | 基本となるコンストラクタ及び、<br>キューサイズを設定する。 |
| DBValidateCollector<P>(QueryRowHandleDAO, String, Object, int, Validator, ValidationErrorHandler) | 基本となるコンストラクタ及び、<br>キューサイズ、<br>使用する入力チェックエラーハンドラクラスを使用する。 |
| DBValidateCollector<P>(QueryRowHandleDAO, String, Object, int, CollectorExceptionHandler, Validator) | 基本となるコンストラクタ及び、<br>キューサイズ、<br>拡張例外ハンドラクラスを設定する。 |
| DBValidateCollector<P>(QueryRowHandleDAO, String, Object, int, CollectorExceptionHandler, Validator, ValidationErrorHandler) | 基本となるコンストラクタ及び、<br>キューサイズ、<br>拡張例外ハンドラクラス、<br>使用する入力チェックエラーハンドラクラスを設定する。 |
| DBValidateCollector<P>(QueryRowHandleDAO, String, Object, int, boolean, CollectorExceptionHandler, Validator) | 基本となるコンストラクタ及び、<br>1:N マッピング使用の有無、<br>キューサイズ、<br>拡張例外ハンドラクラスを設定する。 |
| DBValidateCollector<P>(QueryRowHandleDAO, String, Object, int, boolean, CollectorExceptionHandler, Validator, ValidationErrorHandler) | 基本となるコンストラクタ及び、<br>1:N マッピング使用の有無、<br>キューサイズ、<br>拡張例外ハンドラクラス、<br>使用する入力チェックエラーハンドラクラスを設定する。 |

✧ FileValidateCollector のコンストラクタ一覧

| コンストラクタ | 概要 |
|---|---|
| FileValidateCollector<P>(FileQueryDAO, String, Class<P>, Validator) | 実装例で掲載した基本となるコンストラクタ<br>これら４つの引数は必須である。 |
| FileValidateCollector<P>(FileQueryDAO, String, Class<P>, Validator, ValidationErrorHandler) | 基本となるコンストラクタ及び、<br>使用する入力チェックエラーハンドラクラスを設定する。 |
| FileValidateCollector<P>(FileQueryDAO, String, Class<P>, CollectorExceptionHandler, Validator) | 基本となるコンストラクタ及び、<br>拡張例外ハンドラクラスを設定する。 |
| FileValidateCollector<P>(FileQueryDAO, String, Class<P>, CollectorExceptionHandler, Validator, ValidationErrorHandler) | 基本となるコンストラクタ及び、<br>拡張例外ハンドラクラス、<br>使用する入力チェックエラーハンドラクラスを設定する。 |
| FileValidateCollector<P>(FileQueryDAO, String, Class<P>, int, CollectorExceptionHandler, Validator) | 基本となるコンストラクタ及び、<br>キューサイズ、<br>拡張例外ハンドラクラスを設定する。 |
| FileValidateCollector<P>(FileQueryDAO, String, Class<P>, int, CollectorExceptionHandler, Validator, ValidationErrorHandler) | 基本となるコンストラクタ及び、<br>キューサイズ、<br>拡張例外ハンドラクラス、<br>使用する入力チェックエラーハンドラクラスを設定する。 |

➢ コンストラクタ引数一覧

前ページで列挙したコンストラクタで使用される引数を以下に列挙する。
Collector 機能からの新規要素については**太字**で掲載する

✧ DBValidateCollector のコンストラクタで渡される引数

| 引数 | 解説 | デフォルト値 | 省略 |
|---|---|---|---|
| QueryRowHandleDAO | DB にアクセスするための DAO | ― | 不可 |
| String | SqlMap で定義した SQLID | ― | 不可 |
| Object | SQL にバインドされる値を格納したオブジェクト、バインドする値が存在しない場合は省略せず、null を渡す事。 | ― | 不可 |
| int | キューサイズ、0 以下の値は無視される。基本的に変更不要。 | 20 | 可 |
| CollectorExceptionHandler | 例外ハンドラクラス、 | Null | 可 |
| boolean | iBATIS の 1:N マッピング使用時は true を渡す。<br>true にする事により、メモリの肥大化を最小限に抑えることができる。 | false | 可 |
| **Validator** | **入力チェックを行う Validator。<br>通常は Spring が提供する BeanValidator(Validator インタフェース実装クラス)を使用する。** | ― | **不可** |
| **ValidationErrorHandler** | **入力チェックエラーハンドラクラス。** | **ExceptionValidationErrorHandler** | **可** |

✧ FileValidateCollector のコンストラクタで渡される引数

| 引数 | 解説 | デフォルト値 | 省略 |
|---|---|---|---|
| FileQueryDAO | ファイルにアクセスするための DAO | ― | 不可 |
| String | 読み込むファイル名 | ― | 不可 |
| Class<P> | ファイル行オブジェクトクラス | ― | 不可 |
| int | キューサイズ、0 以下の値は無視される。基本的に変更不要。 | 20 | 可 |
| CollectorExceptionHandler | 例外ハンドラクラス、 | Null | 可 |
| **Validator** | **入力チェックを行う Validator。<br>通常は Spring が提供する BeanValidator(Validator インタフェース実装クラス)を使用する。** | ― | **不可** |
| **ValidationErrorHandler** | **入力チェックエラーハンドラクラス。** | **ExceptionValidationErrorHandler** | **可** |

## 拡張ポイント

- 拡張入力チェックエラーハンドラクラスを独自実装する方法
  - ValidationErrorHandler インタフェースの実装クラスを作成することにより、拡張入力チェックエラーハンドラクラスを作成する事が可能である。
  - 拡張入力チェックエラーハンドラクラスは、以降の処理を制御するステータス ValidateErrorStatus を返却する必要がある。
  - ValidateErrorStatus の一覧表
    (入力チェックエラーハンドリングクラスが返却するステータス)

| ValidateErrorStatus | ビジネスロジックでの next メソッド呼び出し時の挙動 |
|---|---|
| **SKIP** | 入力チェックエラーが発生した場合、その**エラーデータは取得せずに**、その後の処理を継続する。 |
| **CONTINUE** | 入力チェックエラーが発生した場合、その**エラーデータを取得して**、その後の処理を継続する。 |
| **END** | 入力チェックエラーが発生した時点で以降の処理も含めて**強制終了**する。 |

  - 以下に拡張入力チェックエラーハンドラクラスの実装例を掲載する。
    実装例では拡張入力チェックエラーハンドラクラスは以下の仕様で作成する。
    【仕様】
    ① 入力チェックエラー発生時にログレベル info でエラー発生を通知する。
    ② 入力チェックエラーが発生したデータは無視し、以降の処理を継続する。
  - 拡張入力チェックエラーハンドラクラス実装例

```
public class CustomValidationErrorHandler implements ValidationErrorHandler {

    private static Log logger =
        LogFactory.getLog(CustomValidationErrorHandler.class);

    @Override
    public ValidateErrorStatus handleValidationError(
            DataValueObject dataValueObject, Errors errors) {

        // ログ出力
        if(logger.isInfoEnabled()){
            logger.info("入力チェックエラー発生");
        }

        // ValidateErrorStatus の設定
        return ValidateErrorStatus.SKIP;
    }
}
```

ValidationErrorHandler インタフェースを実装する。

handleValidationError メソッドの実装を行う

仕様①に従い、info レベルで
エラーの発生を通知する

仕様②に従い、SKIP を返却することにより、
エラーが発生したデータを無視して、
その後の処理を継続する。

➢ ビジネスロジックの実装例(DB)
(TERASOLUNA Batch Framework for Java ver 3.x の場合)

```
@Component
public class Sample03BLogic extends AbstractTransactionBLogic {

    @Autowired
    protected QueryRowHandleDAO queryRowHandleDAO;

    @Autowired
    private Validator validator;

    CustomValidationErrorHandler handler = new CustomValidationErrorHandler()

    @Override
    public int doMain(BLogicParam param) {

        // Collector の生成
        Collector<Sample03Bean> collector = new DBValidateCollector<Sample03Bean>(
                this.queryRowHandleDAO, "Sample.selectData06", null,
                validator, handler);

        try {
            Sample03Bean inputData = null;
            while (collector.hasNext()) {
                // データの取得
                inputData = collector.next();
                // ここにファイルの出力など、取得データに対する処理を記述する
            }
        } catch (Exception e) {
            // 例外処理
        } finally {
            // Collector の破棄
            CollectorUtility.closeQuietly(collector);
        }
    return 0;
    }
}
```

BeanValidator の DI を行う

独自実装した拡張入力チェックエラーハンドラクラスのインスタンスを生成する。

コレクタ生成時に上で生成した拡張入力チェックエラーハンドラクラスを渡しておく。

collector.next メソッドを呼び出すと、内部で入力チェックが実施される。

このように Collector インスタンス生成時にあらかじめ拡張入力チェックエラーハンドラクラスを渡すことにより、入力チェックエラー発生時にはこのハンドラクラスが使用されることになる。

# ■ リファレンス

## ◆ 構成クラス

| | クラス名 | 概要 |
|---|---|---|
| 1 | jp.terasoluna.fw.collector.db.DBValidateCollector | DBCollector 拡張クラス<br>DBCollector を入力チェックに対応させている。 |
| 2 | jp.terasoluna.fw.collector.file.FileValidateCollector | FileCollector 拡張クラス<br>FileCollector を入力チェックに対応させている。 |
| 3 | jp.terasoluna.fw.collector.validate.ValidationErrorHandler | 入力チェックエラーハンドラインタフェース<br>入力チェックエラーが発生した際の処理を宣言している。 |
| 4 | jp.terasoluna.fw.collector.validate.AbstractValidationErrorHandler | ValidationErrorHandler クラスを実装した抽象クラス<br>コンストラクタによるログレベルの変更や、ログ出力用のメソッドなどの処理を定義している。 |
| 5 | jp.terasoluna.fw.collector.validate.ExceptionValidationErrorHandler | SkipValidationErrorHandler クラスの拡張クラス<br>入力チェックエラーが発生した場合は TRACE ログにエラーコードを出力し、例外をスローする(処理が途中で中断する) |
| 6 | jp.terasoluna.fw.collector.validate.ValidateErrorStatus | 列挙型クラス<br>入力チェックエラーハンドラクラスはこの値によって、入力チェックエラー発生後の挙動を決定する。 |
| 7 | jp.terasoluna.fw.collector.validate.ValidationErrorException | RuntimeException を拡張した入力チェックエラークラス<br>入力チェックエラー発生時にスローされる。 |

## ◆ 関連機能

- 『AL041 入力データ取得機能』

## ◆ 使用例

- **機能網羅サンプル**(terasoluna-batch-functionsample)
- **チュートリアル**(terasoluna-batch-tutorial)

## ◆ 備考

- なし