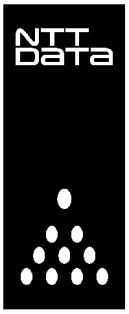


TERASOLUNA Batch Framework for Java

機能説明書

第 3.0.0 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 ver3.0.0 に対応しています。

TERASOLUNA Batch Framework for Java ver3.0.0

■ 概要

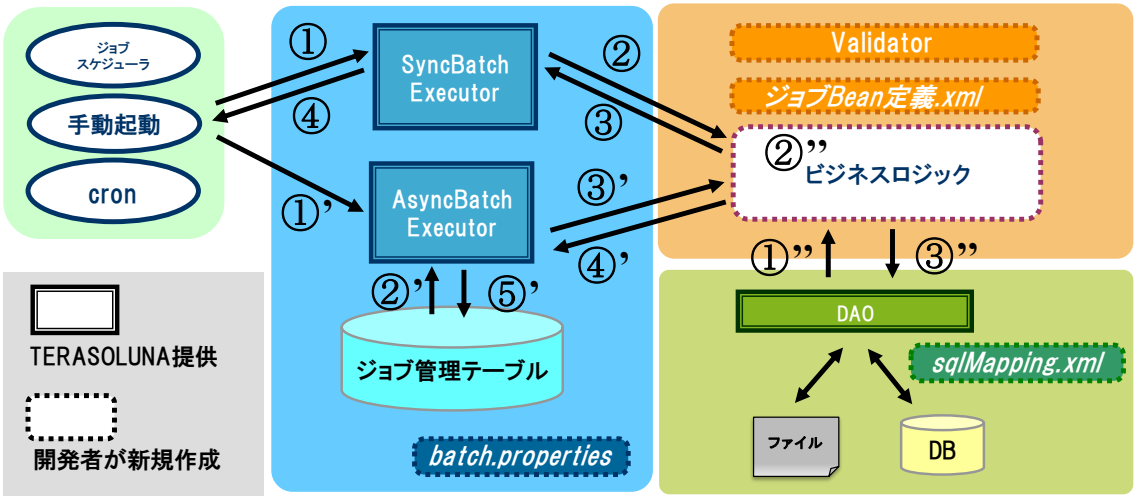
◆ アーキテクチャ概要

- TERASOLUNA Batch Framework for Java ver3.0.0（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

- 同期型ジョブ実行機能 **→BL01参照**
  - 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。
- 非同期型ジョブ実行機能 **→BL02参照**
  - 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。
- トランザクション管理機能 **→BL03参照**
  - 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。
- 例外ハンドリング機能 **→BL04参照**
  - 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。
- ビジネスロジックの実行方式 **→BL01, BL03, BL04参照**
  - 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
- 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。

◆ 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	2.0.3.0
terasoluna-dao.jar	DAO インタフェースを提供する	2.0.3.0
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	2.0.3.0
terasoluna-validator.jar	ファイルアクセス機能を提供する	2.0.3.0
terasoluna-filedao.jar	入力チェック機能を提供する	2.0.3.0

- 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
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

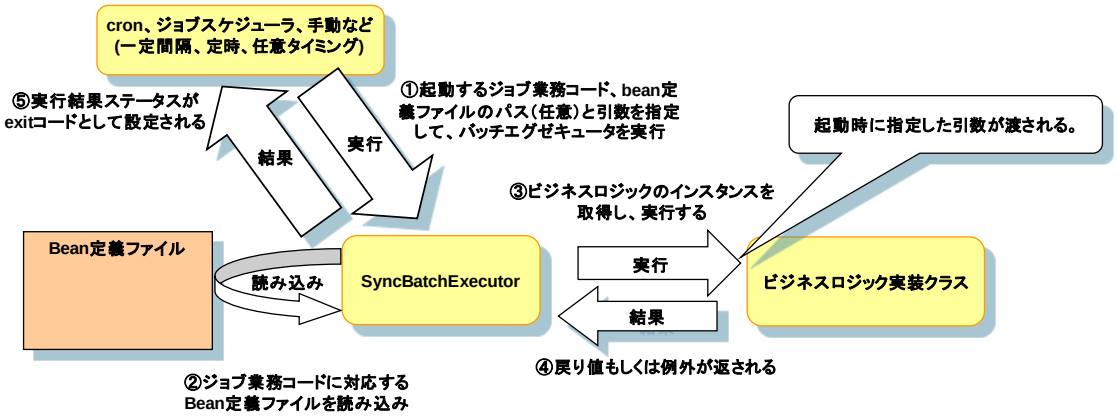
BL01 同期型ジョブ実行機能

■ 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. # 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-2.0.3.0.jar"
sh_classpath=${sh_classpath}:${lib_path}/terasoluna-validator-2.0.3.0.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">
  <context:include-filter type="assignable"
    expression="jp.terasoluna.fw.batch.blogic.BLogic"/>
</context:component-scan>
・・・
```

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。

BLogic インタフェースの実装クラスが自動的にロードされる。

- 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）

- 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）

```
public class B000001BLogic implements BLogic {

  @Autowired
  private QueryDAO queryDAO = null;

  public int execute(BLogicParam param) {

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

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）

```
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。 指定のジョブ業務を実行する。

■ 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』
- 『BC-02 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』

■ 備考

なし

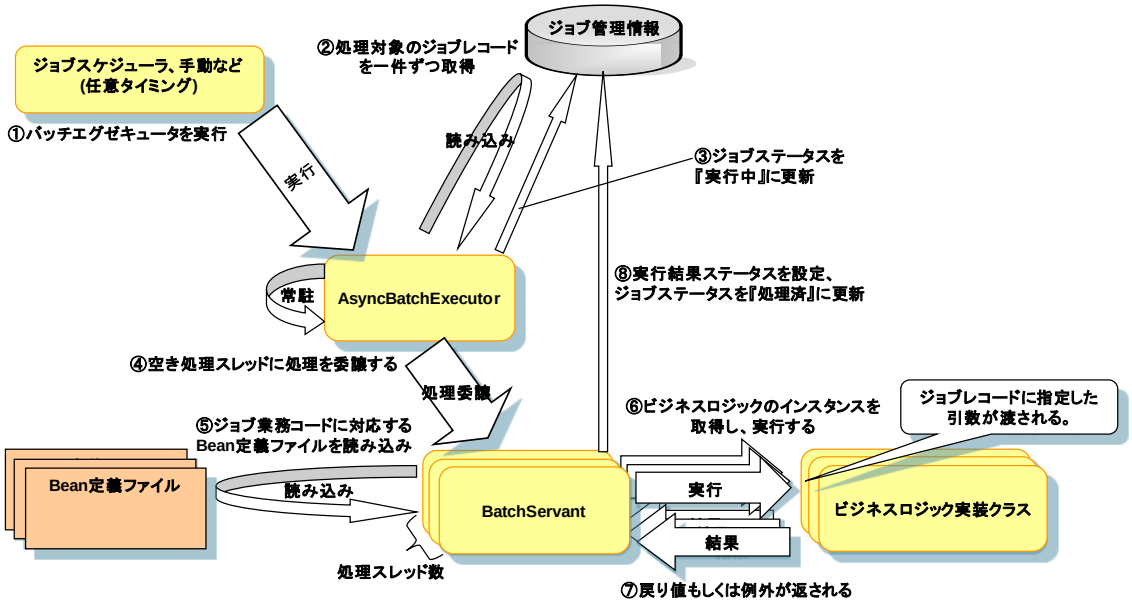
BL02 非同期型ジョブ実行機能

■ 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 ジョブのステータスは以下の 3 つとなる。 未実施：0 実行中：1 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. # 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 ※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 ※ジョブ管理情報テーブルを k 更新する
systemDataSource.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

- 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のチェックしている。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、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

■ 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-2.0.3.0.jar"
sh_classpath=${sh_classpath}:${lib_path}/terasoluna-validator-2.0.3.0.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。 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。 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。 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

■ 備考

なし

BL03 トランザクション管理機能

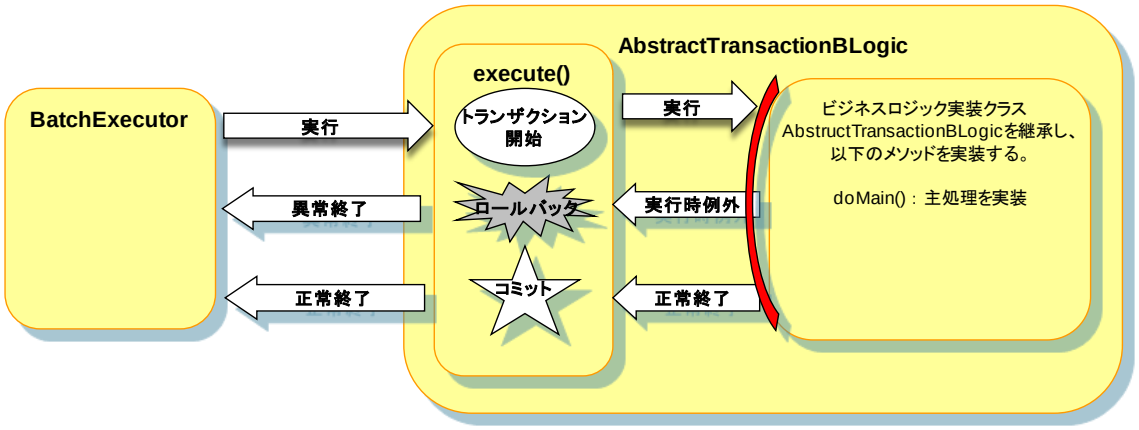
■ 概要

◆ 機能概要

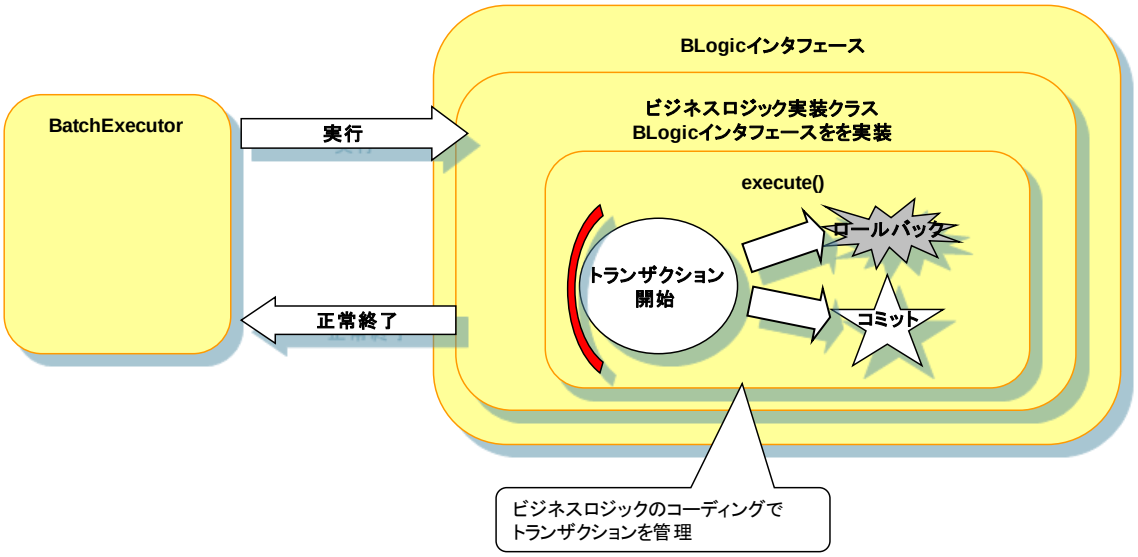
- フレームワークで以下の二つのトランザクションモデルを提供する。
  開発者は、業務要件に応じてトランザクションモデルを選択する。
  - ➢ フレームワークがトランザクションを管理するモデル
    - ✧ 1 ビジネスロジック 1 トランザクションで完結するモデル.。通常はこちらのモデルを選択する。AbstractTransactionBLogic を継承する
  - ➢ ビジネスロジックで任意にトランザクションを管理するモデル
    - ✧ 複雑なトランザクション管理を必要とする場合に選択する。BLogic インタフェースを実装する

◆ 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。

```
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、トランザクション管理を行う。

```
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。

```
public class B000002BLogic implements BLogic {
    @Autowired
    private PlatformTransactionManager transactionManager = null;

    @Override
    public int execute(BLogicParam param) {
```

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

```
            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);
          }
      }
}
```

■ 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。 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。 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。 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

- 『BB-01 データベースアクセス機能』

■ 備考

- 大量データを扱うジョブの実装について

  大量のデータを DB やファイルから読み込み処理をする場合、通常の QueryDAO を利用すると大量データを一括で取得し、バッチサーバ上のメモリが枯渇することで OutOfMemoryError が発生する可能性がある。その場合はフレームワークが提供する queryRowHandleDAO を利用する。

  queryRowHandleDAO を利用することで、大量データを 1 件ずつ取得して処理することが可能である。

  実装するには、ビジネスロジック内で queryRowHandleDAO を利用する。また DateRowHandle インタフェースを実装したクラスで 1 件ずつ処理を行う。

  - データソース Bean 定義ファイルの設定例

```
<!-- 照会系の QueryRowHandleDAO 定義 -->
<bean id="queryRowHandleDAO"
      class="jp.terasoluna.fw.dao.ibatis.QueryRowHandleDAOiBatisImpl">
      <property name="sqlMapClient" ref="sqlMapConfig"/>
```

```
</bean>
```

- 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.b000001">
   <context:include-filter type="assignable"
                           expression="jp.terasoluna.fw.batch.blogic.BLogic"/>
   <context:include-filter type="assignable"
                           expression="jp.terasoluna.fw.dao.event.DataRowHandler"/>
</context:component-scan>
```

- ビジネスロジックの実装例

```
@Autowired
private QueryRowHandleDAO queryRowHandleDAO = null;

public int doMain(BLogicParam param) {
        // SQL-ID
        String sqlID = "b000001.selectUser";
        // SQL にバインドする値
        Object bindParams = null;
        // SQL 実行
        queryRowHandleDAO.executeWithRowHandler(sqlID, bindParams,
                userDataRowHandler);
        return  BATCH_NORMAL_END;
}
```

- RowHandler 実装クラスの実装例

```
public class UserDataRowHandler implements DataRowHandler {

   @Autowired
   private UpdateDAO updateDAO = null;

   public void handleRow(Object arg) {
      if (arg instanceof SelectUser) {
                SelectUser su = (SelectUser) arg;
                updateDAO.execute("b000001.insertUser", su);
      }
   }
}
```

※RowHandler 実装内で更新系の処理を行う場合は、AbstractTransactionBLogic を利用する場合はビジネスロジック終了後に一括でコミットされる。もし何件毎にコミットしたい場合、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を利用し、REQUIRED_NEW でトランザクションを管理するか、更新系と参照系コ

ネクションは別のコネクションを利用するようにすること。複数のコネクションを扱いたい場合は下記の「複数データソースを利用について」を参照する。

- 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。

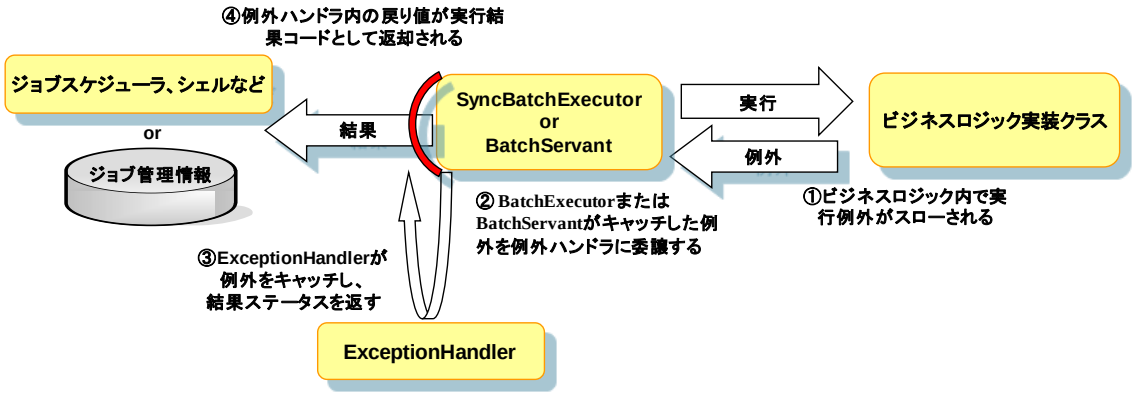
BL04 例外ハンドリング機能

■ 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

```
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。

- Bean 定義ファイルの設定
  - コンポーネントスキャンの設定に ExceptionHandler の定義を追加する。
  - その他について同期型ジョブ実行機能を参照。

```
…略…
<!-- コンポーネントスキャン設定 -->
<context:component-scan base-package="jp.terasoluna.batch.sample.b000001">
    <context:include-filter type="assignable"
    expression="jp.terasoluna.fw.batch.blogic.BLogic"/>
    <context:include-filtertype="assignable"
    expression="jp.terasoluna.fw.batch.exception.handler.ExceptionHandler"/>
</context:component-scan>
…略…
```

コンポーネントスキャンの設定に ExceptionHandler 定義を追加する。

■ 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。 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。 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

■ 備考

なし

バッチ更新最適化機能

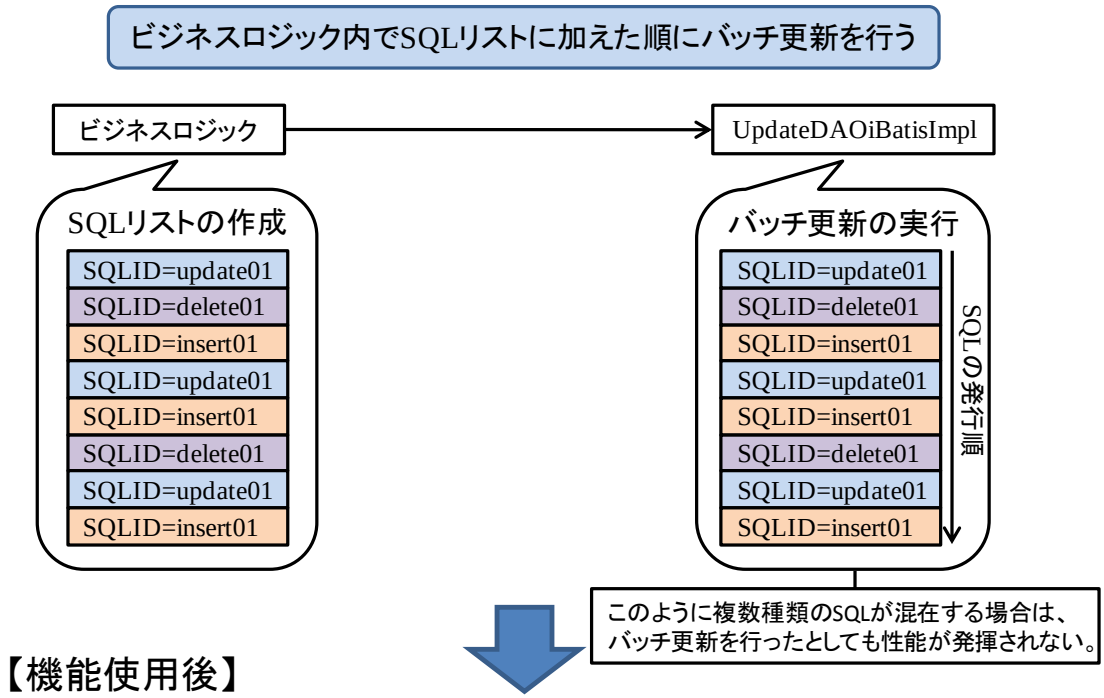
■ 概要

◆ 機能概要

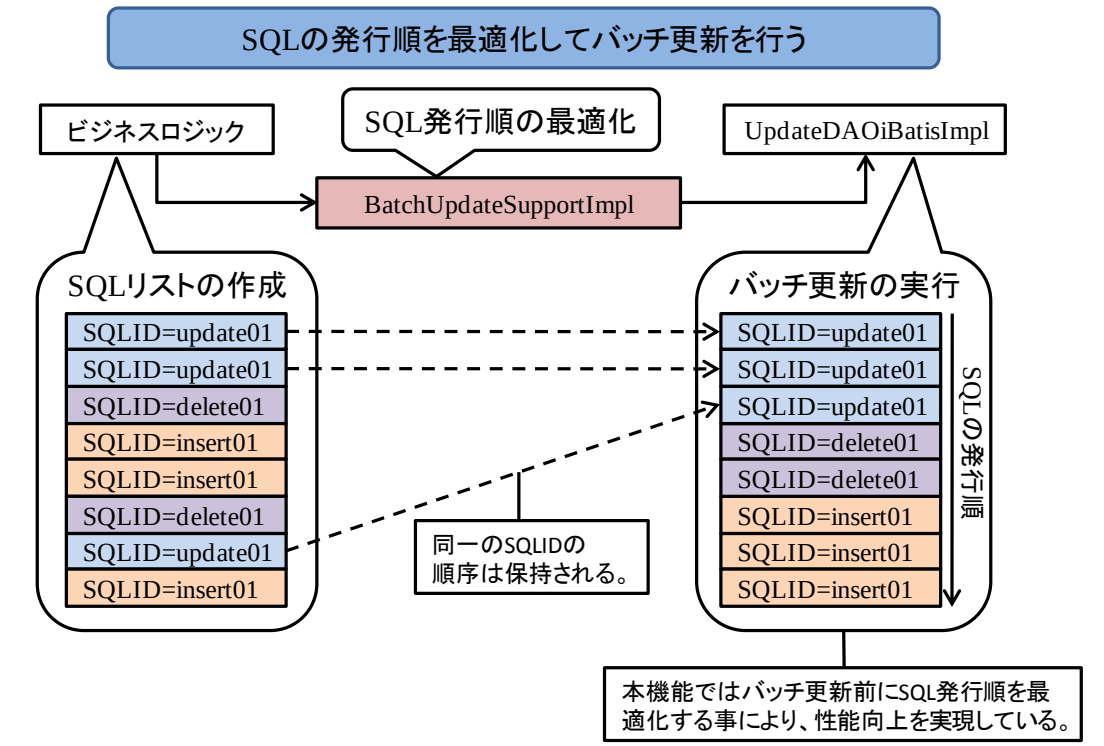
- バッチ更新を行う前に SQL の発行順を最適化する機能を提供する。
- SQL の最適化を行うと、バッチ更新時に発行する SQL の種類が 2 種類以上の場合に、本機能による性能の向上が期待できる。
  - （発行する SQL が単一の場合は、従来のバッチ更新との差は無い）
- 最適化により SQL の発行順が変更される為、「AL-036 バッチ更新最適化機能」を使用する際は、SQL の発行順が変更されても問題が無い事が前提条件となる。

◆ 概念図

【機能使用前】



【機能使用後】



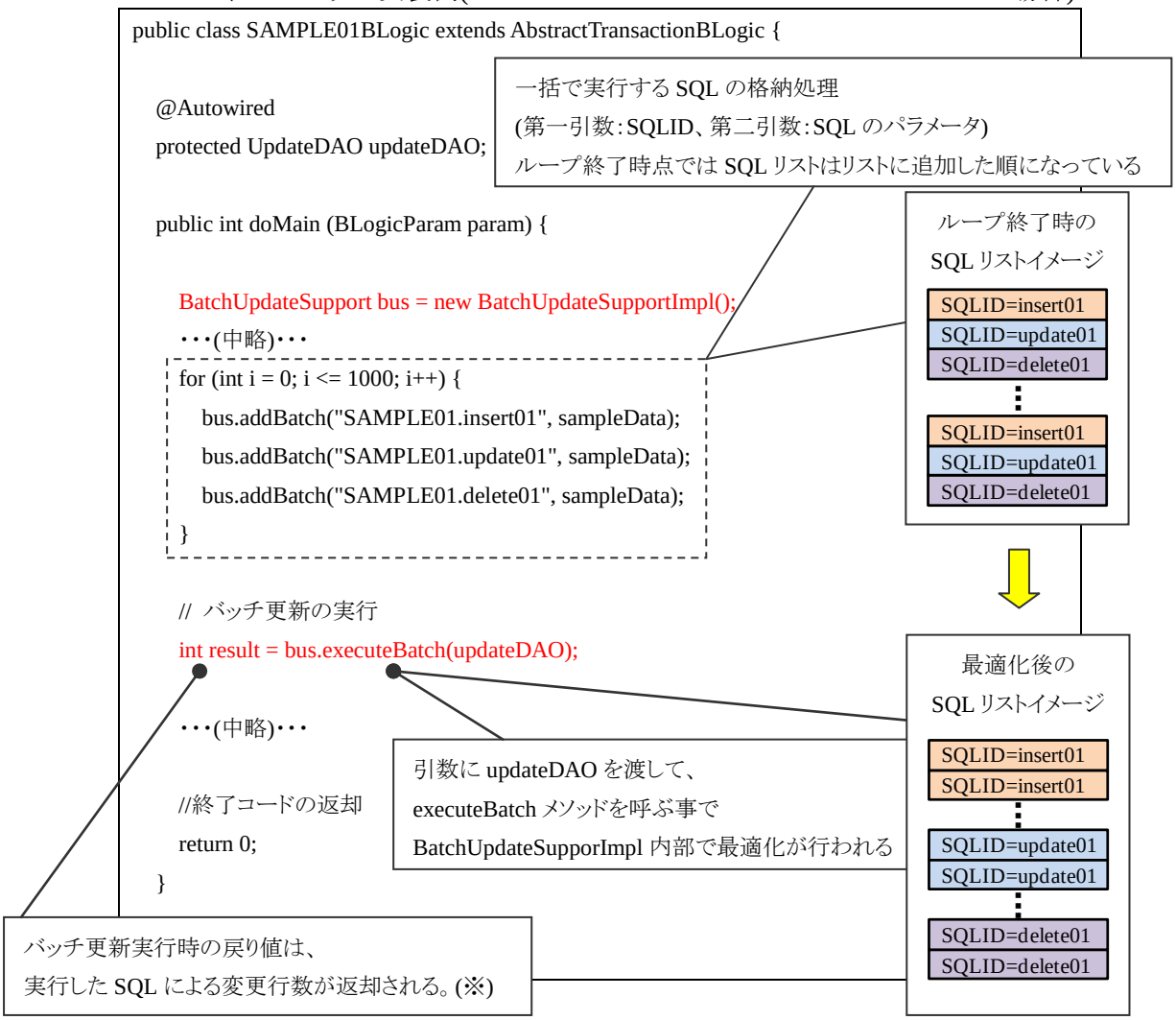
◆ 解説

- 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

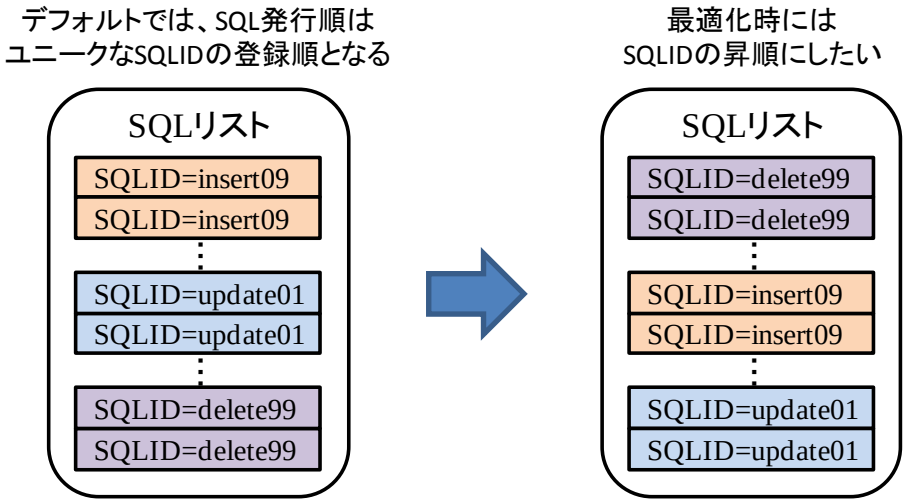
- 本機能を使用する際の実装例
  - ビジネスロジック実装例(TERASOLUNA Batch Framework for Java 3.x の場合)



※ 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);
・・・
```

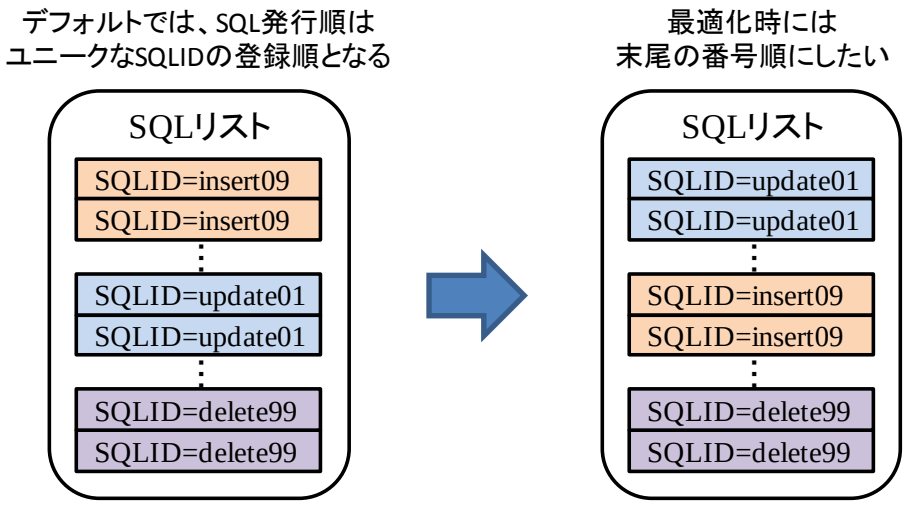
バッチ更新実行前に、sort メソッドを呼び出す事によって、内部的にソートフラグが立つ。

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。

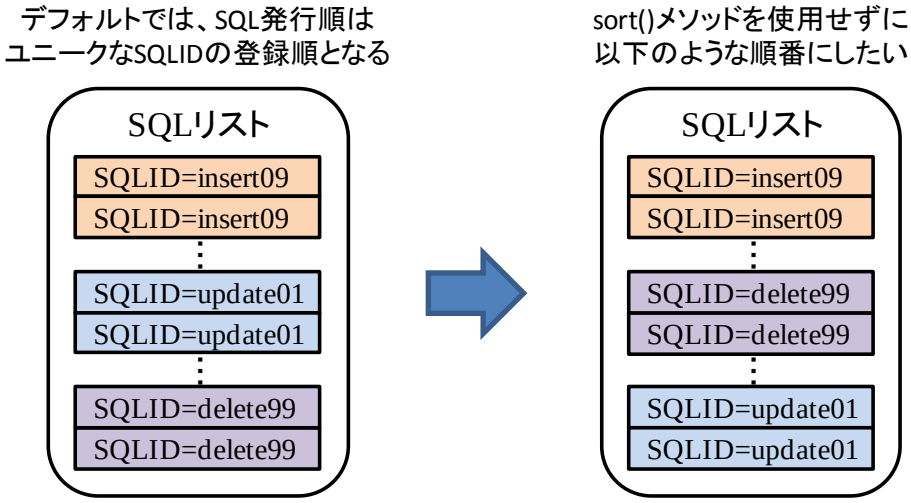
- ビジネスロジッククラスの実装例(部分的に抜粋)

```
・・・
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。 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。 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。 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。 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。