TERASOLUNA Batch Framework for Java ver 3.5.1

■ 概要

アーキテクチャ概要

- TERASOLUNA Batch Framework for Java ver 3.5.1（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。
- 本フレームワークはSpring Framework、MyBatis3をベースフレームワークとしている。

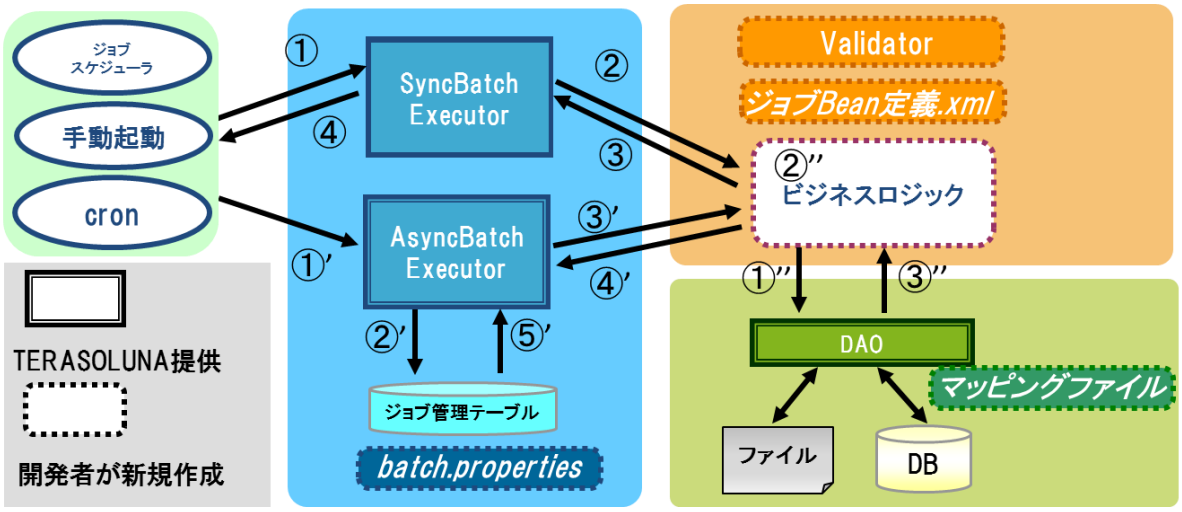
◆ 機能概要

- 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-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件ごと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
  - Java SE7.0/8.0
- 対応データベース
  - Oracle 12c
  - PostgreSQL 9.3.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-batch	同期型バッチ実行機能、非同期型バッチ実行機能、トランザクション管理機能、ビジネスロジック実行機能、メッセージ管理機能を提供する	3.5.1
terasoluna-logger	汎用ログ・汎用例外メッセージログ出力機能を提供する	3.5.1
terasoluna-filedao	ファイルアクセス機能を提供する	3.5.1
terasoluna-collector	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	3.5.1
terasoluna-commons	ユーティリティ機能など共通機能を提供する	3.5.1

● 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
aopalliance	1.0
aspectjweaver	1.8.4
classmate	1.0.0
commons-beanutils	1.9.2
commons-collections	3.2.1
commons-dbcp2	2.0.1
commons-digester	2.1
commons-jxpath	1.3
commons-lang3	3.3.2
commons-pool2	2.2
hibernate-validator	5.1.3.Final
javax.el	3.0.0
javax.inject	1
jboss-logging	3.1.3.GA
jcl-over-slf4j	1.7.8
logback-classic	1.1.2
logback-core	1.1.2
mybatis	3.2.8
mybatis-spring	1.2.2
oro	2.0.8
slf4j-api	1.7.8
spring-aop	4.1.4.RELEASE
spring-beans	4.1.4.RELEASE
spring-context	4.1.4.RELEASE
spring-core	4.1.4.RELEASE
spring-expression	4.1.4.RELEASE
spring-jdbc	4.1.4.RELEASE
spring-tx	4.1.4.RELEASE
validation-api	1.1.0.Final

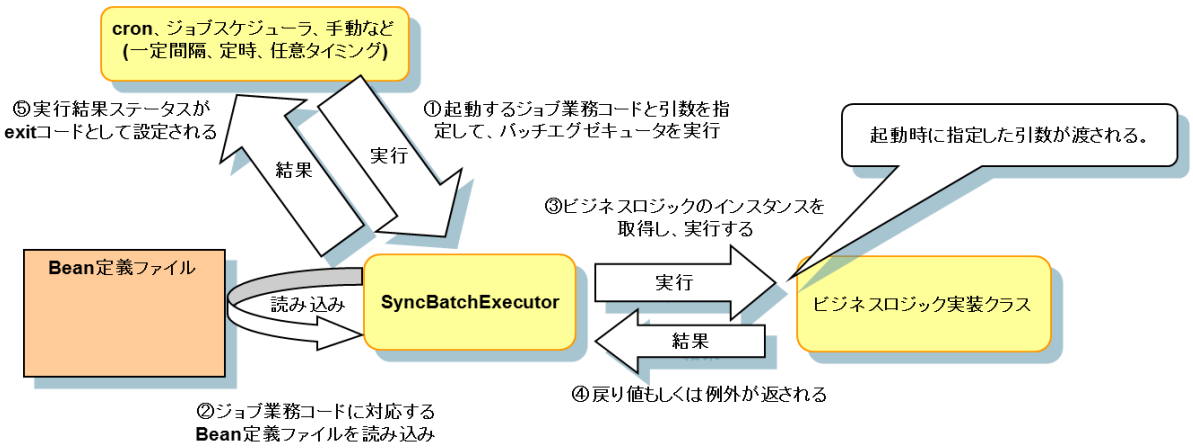
BL-01 同期型ジョブ実行機能

■ 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み込まれる。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	beansAdminDef/	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 での設定例をもとに説明する。

- クラスパスファイル（classpath.sh）の設定を行う。

```
export CLASSPATH=../lib/*
```

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

```
#!/bin/sh


# 共通 CLASSPATH 定義シェル実行
. ./classpath.sh

# バッチ起動
java jp.terasoluna.fw.batch.executor.SyncBatchExecutor B000001 2 3 4
```

ジョブ業務コードを第 1 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
  - ジョブ業務コード：B000001
  - 実行パラメータ：[2, 3, 4]

```
#!/bin/sh


# 共通 CLASSPATH 定義シェル実行
. ./classpath.sh

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

# バッチ起動
java jp.terasoluna.fw.batch.executor.SyncBatchExecutor
```

ジョブ業務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

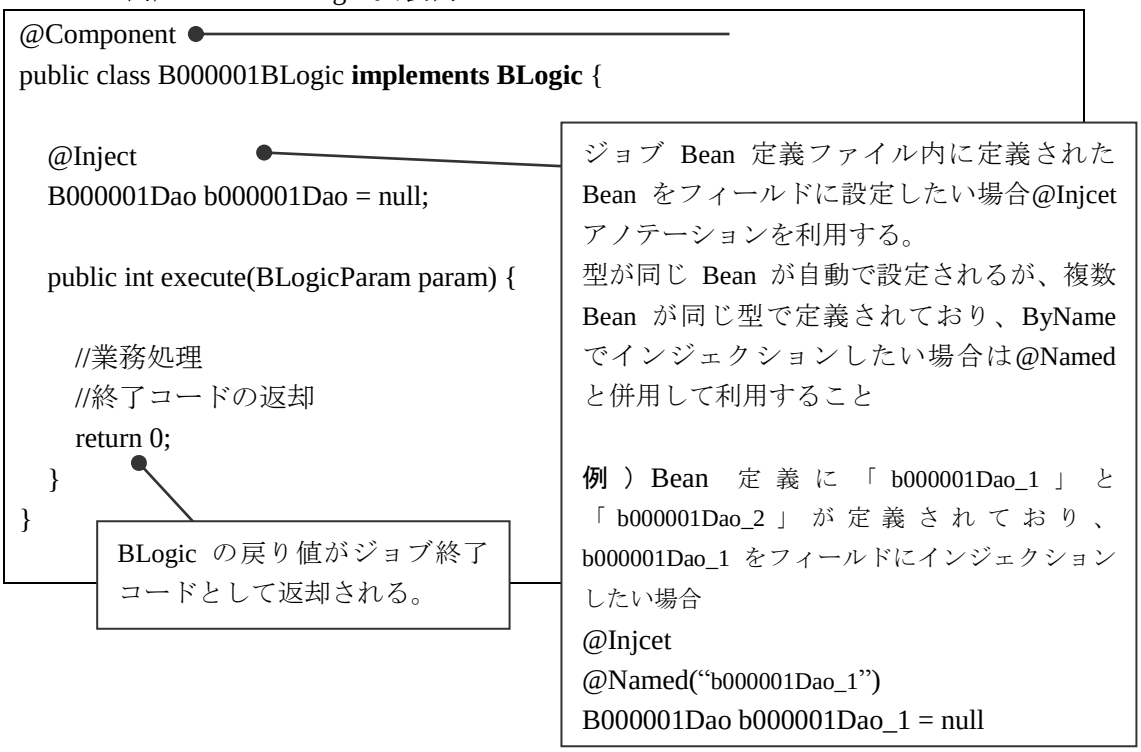
```
・・・
<!-- アノテーションによる設定 -->
<context:annotation-config/>

<!-- 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。

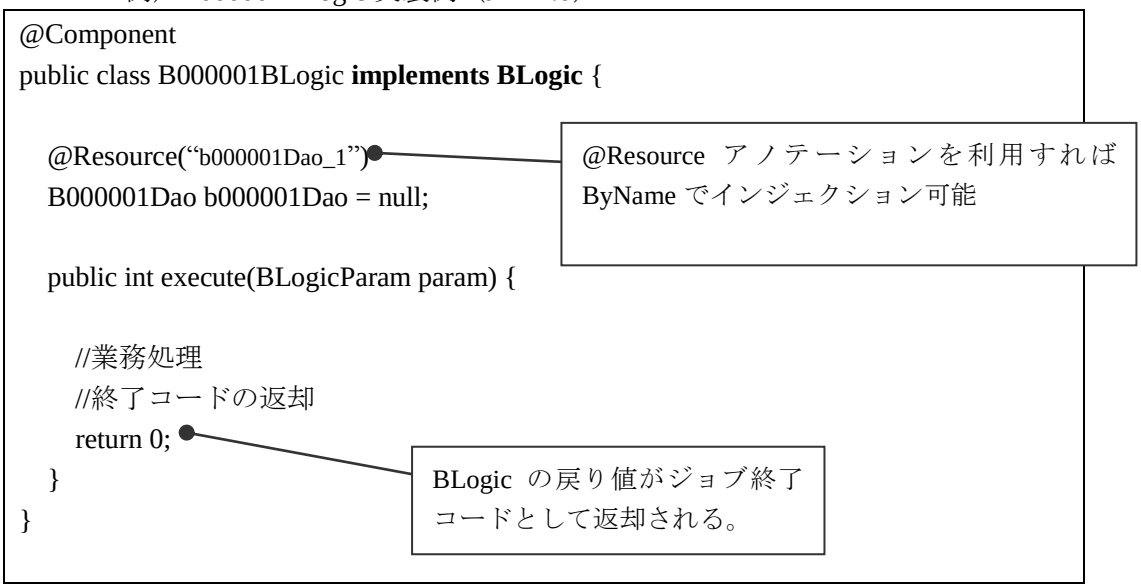
- ビジネスロジックの実装
  - BLogic インタフェースを実装する
    - トランザクションをフレームワーク側で管理する場合は抽象クラスである AbstractTransactionBLogic クラスを継承すること。(詳細は後述するトランザクション管理機能を参照すること）
  - クラス名の宣言に@Componet アノテーションを付与し、先のコンポーネントスキャンの対象にする。
  - Inject アノテーションを利用して Bean 定義ファイルに定義した Bean をフィールドにインジェクションできる。
  - execute メソッドを実装する。
    - 引数として渡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

@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。

例）B000001BLogic 実装例



例）B000001BLogic 実装例（JDK7.0）



◆ 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

- 『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ことにより、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。

  - 使用例
    以下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。

例）B000001BLogic 実装例

```
@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。

  - 注意点
    @JobComponent 機能を有効化するには batch.properties ファイルに以下のように設定すること。

例）batch.properties 設定例

```
enableJobComponentAnnotation=true
```

- 異常時のリカバリについて

  同期型ジョブ実行機能には異常時にリカバリを行うための仕組みが備わっていない。異常時のリカバリ（検知と再実行）の仕組みはアプリケーションで実装する必要がある。以下に一例を示す。

  - ジョブの異常を検知する

    ジョブの実行中は定期的にログを出力するようにビジネスロジックを実装しておき、ジョブスケジューラ等で実行中のジョブのログが出力されていることを確認する。

  - 同期型バッチエグゼキュータを強制終了する

    ジョブスケジューラの機能などで検知した異常ジョブを実行している同期型バッチエグゼキュータのプロセスを停止させる。

  - ジョブを再実行する

    異常終了したジョブに対する影響調査を行った後に、同期型バッチエグゼキュータを起動し、ジョブを再実行する。

    ※異常終了地点からの再開（リスタート）機能がないことを考慮したリカバリ設計をしておくこと。

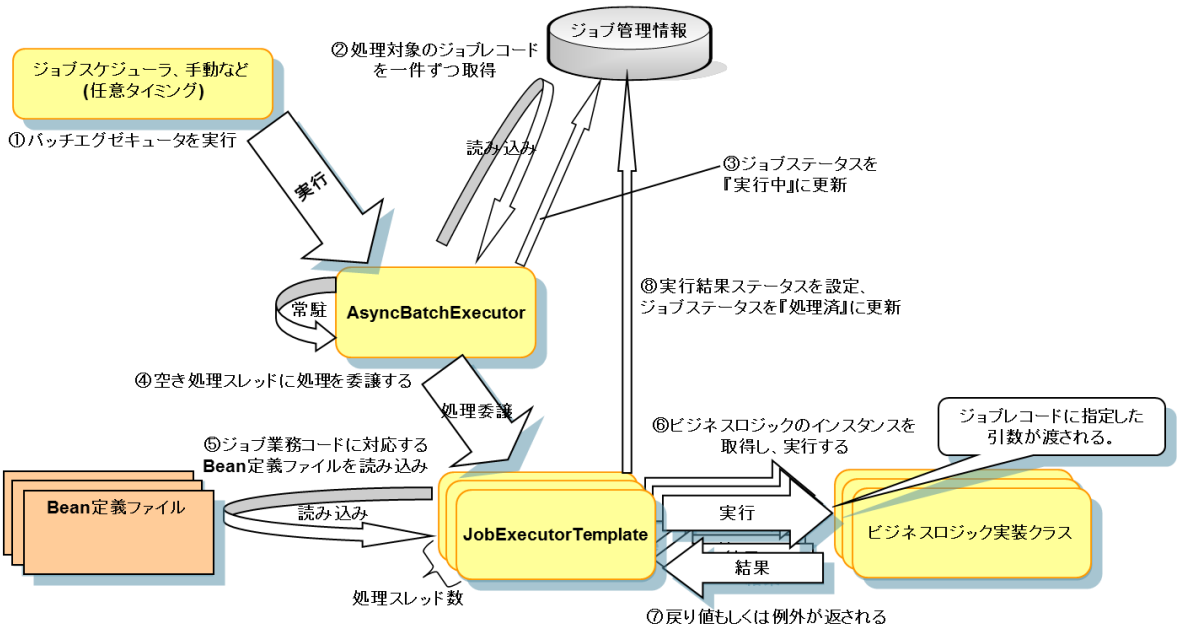
BL-02 非同期型ジョブ実行機能

■ 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ごと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。
    - ThreadPoolTaskExecutor 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="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
      <property name="threadFactory" ref="separateGroupThreadFactory"/>
      <property name="corePoolSize" value="10" />
      <property name="maxPoolSize" value="10" />
      <property name="queueCapacity" value="10" />
</bean>
```

プロパティ	説明
threadFactory	スレッドの生成に使用する Factory クラスを設定する(固定) (※1) スレッドを作成する際に新しいスレッドグループを割り当てる。フレームワークではスレッドグループをキーにアプリケーションコンテキストを管理している。
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 (※2) 設定値は corePoolSize=maxPoolSize にすることを推奨する。フレームワークでは古いスレッドグループを明示的に破棄できないため、設定値が corePoolSize≠maxPoolSize の場合メモリリークの原因になる可能性がある。
queueCapacity	内部キューの容量を指定する。 (※3) コアスレッド数を超えてスレッドが作成されるには追加されるジョブの数がキューの容量を超える必要があることに注意すること。デフォルトの容量は Integer.MAX_VALUE である。 (※4) queueCapacity は maxPoolSize 以上の値を設定しておくことを推奨する。maxPoolSize より小さい値を設定すると、TaskRejectException が発生し、ジョブが実行されない可能性がある。

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_date_time		ジョブ登録時刻
26	更新時刻	upd_date_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み込みまれる。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	beansAdminDef/	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.admin.dataSource	AdminDataSource.xml	管理用 Bean 定義（データソース部）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	メッセージアクセサの Bean 名
systemDataSource.systemDao	systemDao	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. 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）
batchTaskExecutor.availableThreadThresholdCount	100	空きスレッド残数閾値のデフォルト値
batchTaskExecutor.availableThreadThresholdWait	30	空きスレッド残数閾値以下の場合のウェイト時間（ミリ秒）のデフォルト値
batchTaskExecutor.executeRetryInterval	100	ジョブ実行リトライ間隔（ミリ秒）
batchTaskExecutor.executeRetryCountMax	1	ジョブ実行リトライ回数

- 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。

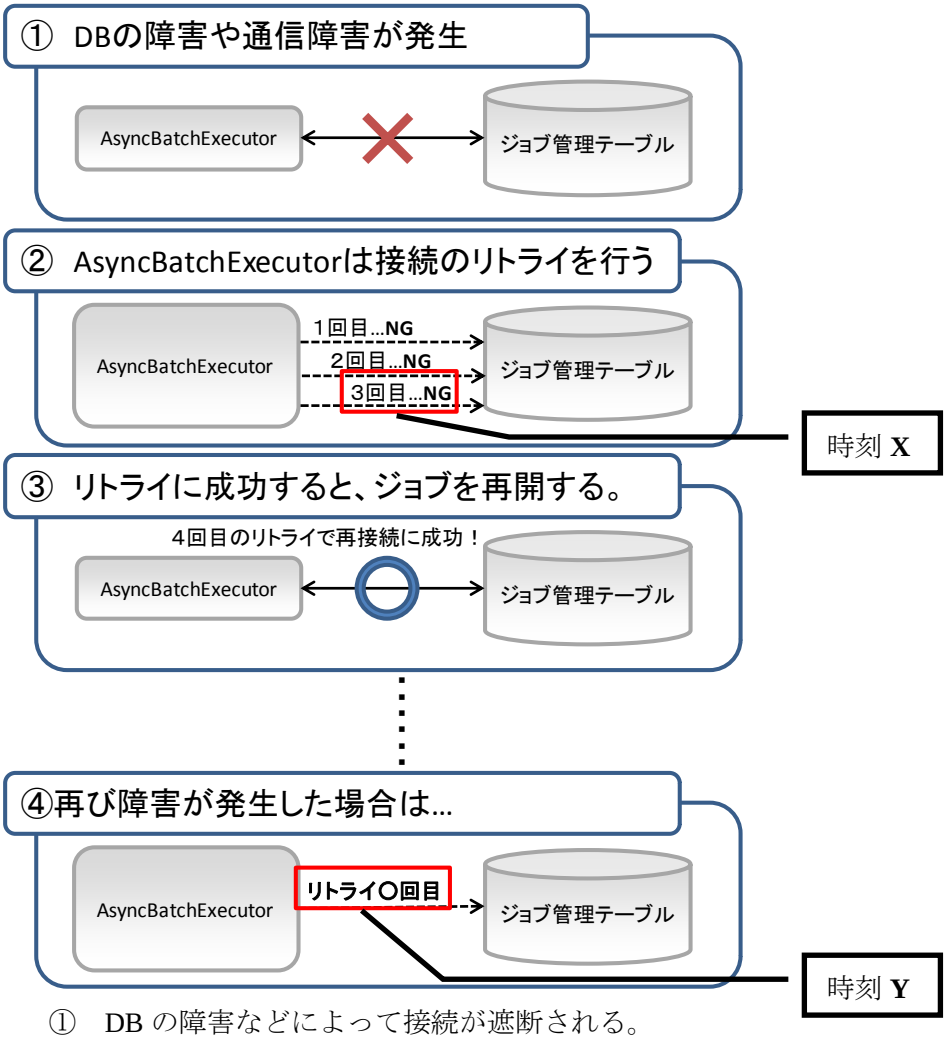
例）executor.endMonitoringFile=/tmp/batch_terminate_file と設定されている場合、windows 環境であれば C:¥tmp フォルダに batch_terminate_file というファイル名を配置すれば、非同期型バッチエグゼキュータは終了する。(※ファイルの内容はなくてもよい）

- ✧ 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ことができる。
  - 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。

- アプリケーション資材入れ替え時の注意点
  - 本機能は常駐プロセス動作中の設定ファイル・ライブラリ等アプリケーション資材の動的な差し替えには対応していない。
  - メンテナンスやライブラリバージョンアップ等に伴うアプリケーション資材の入れ替えを行う場合、常駐プロセスを終了させたうえでアプリケーション資材の入れ替えを実施し、入れ替え完了後に常駐プロセスの再起動を実施すること。

■ 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 での設定例をもとに説明する。

  - クラスパスファイル（classpath.sh）の設定を行う。

```
export CLASSPATH=../lib/*
```

  - 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



# 共通 CLASSPATH 定義シェル実行
. ./classpath.sh

# バッチ起動
java  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。

※ 上記の起動方法ではすべてのジョブが処理対象となるが、ジョブ業務コードを第1引数に設定して起動すると処理対象のジョブを絞ることができる。

- プロパティファイルの設定
  - ジョブ管理テーブルへの接続設定が必要。デフォルトでは mybatisAdmin/jdbc.properties に接続情報を記載する。

- 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

■ 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
  BL-01 同期型ジョブ実行機能と同様に利用可能である。詳しくは BL-01 を参照。

- 異常時のリカバリについて
  非同期型ジョブ実行機能には異常時にリカバリを行うための仕組みが備わっていない。異常時のリカバリ（検知と再実行）の仕組みはアプリケーションで実装する必要がある。以下に一例を示す。

  - ジョブの異常を検知する
    ジョブ管理テーブルの「更新時刻」カラムは、フレームワークがジョブを起動した時刻、または、終了した時刻で更新される。
    この仕組みを利用し、ジョブスケジューラ等で現在時刻と更新時刻の時間をチェックする SQL を定期的に実行し、時間差が一定以上のジョブを異常として検知する。

  - 非同期型バッチエグゼキュータの異常を検知する
    ジョブスケジューラ等で非同期型バッチエグゼキュータプロセスの死活監視を行い、非同期バッチエグゼキュータプロセスが異常終了していた場合は影響調査を行ったうえで、再起動する。

  - 異常ジョブを強制終了する
    フレームワークの機能を使用して、非同期型ジョブ実行機能の実行中に、ジョブ単位に実行中の処理を停止することはできない。ジョブ単位に停止する仕組みはアプリケーションで実装する必要がある。
    ジョブ単位に停止させる方法としては、ビジネスロジック内でジョブごとの終了ファイルによる終了判定や、タイムアウト判定を組み込む方法がある。

  - 非同期型バッチエグゼキュータを強制終了する
    終了ファイルを配置し、異常ジョブ以外のジョブの正常終了を待つ。その後、検知した異常ジョブを実行している非同期型バッチエグゼキュータのプロセスごと停止させる。終了ファイルを配置するだけでは、異常ジョブの終了を待ち続けてしまうため、非同期バッチエグゼキュータが終了することはない。

  - ジョブを再実行する
    影響調査を行ったうえで、ジョブ管理テーブルのジョブステータスを「0：未実施」に更新する。非同期型バッチエグゼキュータが起動後、再実行対象となる。
    ※ジョブを再実行する前に他のジョブを動作させたい場合は、異常終了したジョブのジョブステータスを 0,1,2 以外の値に更新しておき、動作させたいタイミングで「0：未実施」に更新する。

BL-03 トランザクション管理機能

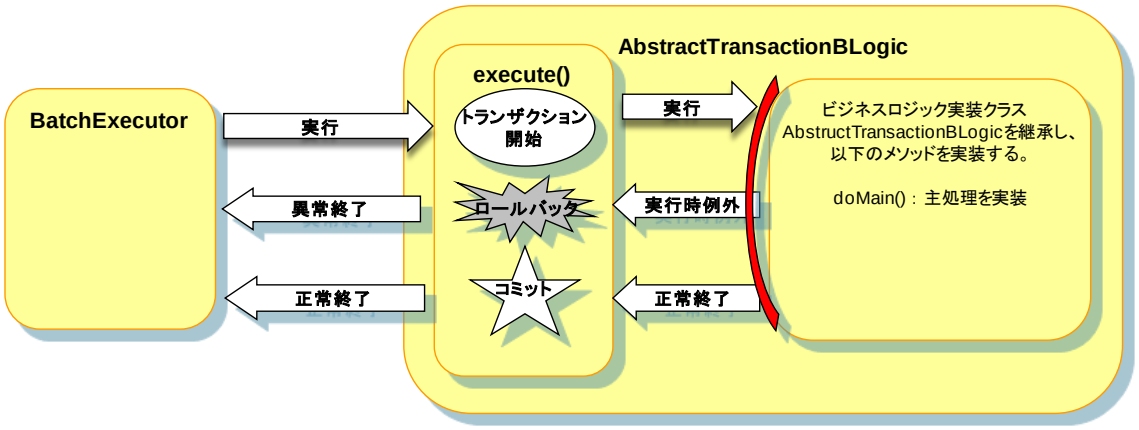
■ 概要

◆ 機能概要

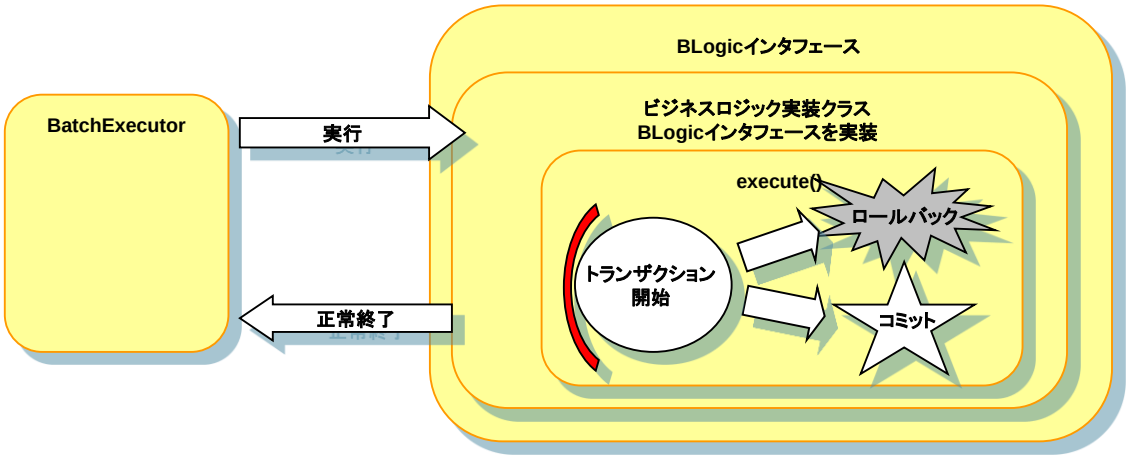
- フレームワークで以下の二つのトランザクションモデルを提供する。
  開発者は、業務要件に応じてトランザクションモデルを選択する。
  - フレームワークがトランザクションを管理するモデル
    - ✧ 1 ビジネスロジック 1 トランザクションで完結するモデル。通常はこちらのモデルを選択する。AbstractTransactionBLogic を継承する
  - ビジネスロジックで任意にトランザクションを管理するモデル
    - ✧ 複雑なトランザクション管理を必要とする場合に選択する。BLogic インタフェースを実装する

◆ 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 {

    @Inject
    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 件ごと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 {

    @Inject
    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。 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

- 『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.dbcp2.BasicDataSource">
        …　略　…
</bean>

<bean  id="transactionManager_1"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
   <property  name="dataSource"  ref="dataSource_1"  />
</bean>

…以下、sqlSessionFactory、sqlSessionTemplate 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.dbcp2.BasicDataSource">
        …　略　…
</bean>

<bean  id="transactionManager_2"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
   <property  name="dataSource"  ref="dataSource_2"  />
</bean>

…以下、sqlSessionFactory、sqlSessionTemplate の Bean 定義を設定する…
```

➢ ジョブ Bean 定義ファイルの設定例

```
<!-- データソース 1 を使用する DAO -->
<bean id=" b000001Dao_1"   class="org.mybatis.spring.mapper.MapperFactoryBean ">
      <property name=”mapperInterface” value=”～.B000001Dao_1” />
      <property name=”sqlSessionFactory” ref=”sqlSessionFactory_1” />
</bean>

<!-- データソース 2 を使用する DAO -->
<bean id=" b000001Dao_2"   class="org.mybatis.spring.mapper.MapperFactoryBean ">
      <property name=”mapperInterface” value=”～.B000001Dao_2” />
      <property name=”sqlSessionFactory” ref=”sqlSessionFactory_2” />
</bean>
```

➢ ビジネスロジックの設定例

```
@Inject
@Named(“b000001Dao_1”)
B000001Dao_1 b000001Dao_1 = null;

@Injcet
@Named(“b000001Dao_2”)
B000001Dao_2 b000001Dao_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。

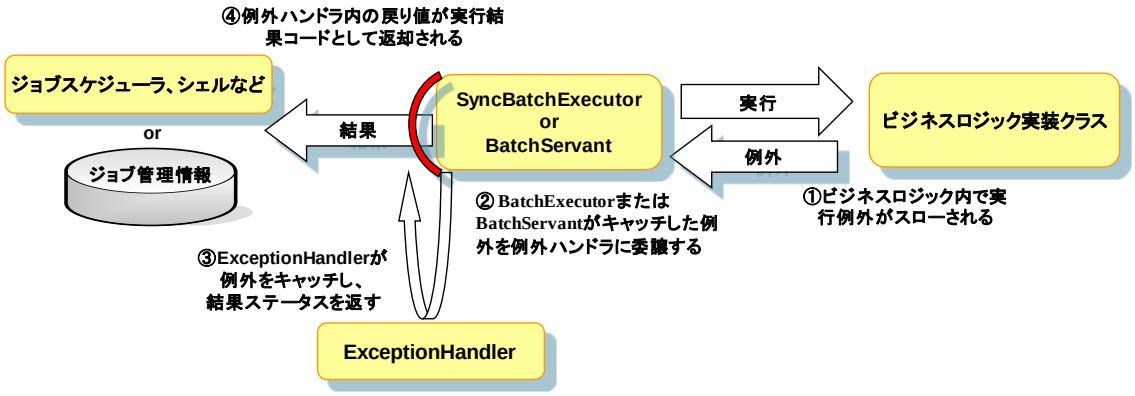
BL-04 例外ハンドリング機能

■ 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

- Bean 定義ファイルの設定
  - 業務処理で例外が発生した場合、DefaultExceptionHandler(デフォルト例外ハンドラ)は、WARN レベルの例外ログを出力し、例外の種類に応じた終了コードへの変換を行う。
  - 例外と終了コードの変換テーブルは Bean 定義ファイルに exceptionToStatusMap として設定する。
  - 例外の型と一致しているかどうかは exceptionToStatusMap の設定順にチェックするため、詳細な例外から順に設定すること。どの例外の型とも一致しない場合は、255 に変換する。

例）例外と終了コードの変換情報の設定例(commonContext.xml)

```
<bean id="defaultExceptionHandler"
        class="jp.terasoluna.fw.batch.exception.handler.DefaultExceptionHandler">
    <property name="exceptionToStatusMap" ref="exceptionToStatusMap"/>
</bean>

<util:map id="exceptionToStatusMap">
    <entry key="org.springframework.dao.RecoverableDataAccessException"
        value="128" />
    <entry key="org.springframework.dao.DataAccessException" value="254" />
</util:map>
```

exceptionToStatusMap に設定を追加する。

RecoverableDataAccessException は DataAccessException のサブクラスなので先に指定している。

- 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した例外ハンドリングクラスを実装し、デフォルト例外ハンドラの処理を置き換えることができる。
  - 例外ハンドリングクラス名は「ジョブ業務コード」+「ExceptionHandler」と命名すること。
  - 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 Logger log = LoggerFactory.getLogger(B000001ExceptionHandler.class);

    @Inject
    MessageAccessor messageAccessor;

    @Override
    public int handleThrowableException(Throwable e
        // WARN ログを出力する
        if (log.isWarnEnabled()) {
            log.warn(messageAccessor.getMessage("errors.exception", null));
            log.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。 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

■ 使用例

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

■ 備考

なし