TERASOLUNA Batch Framework for Java ver 3.6.0

■ 概要

◆ アーキテクチャ概要

- TERASOLUNA Batch Framework for Java ver 3.6.0(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。
- 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

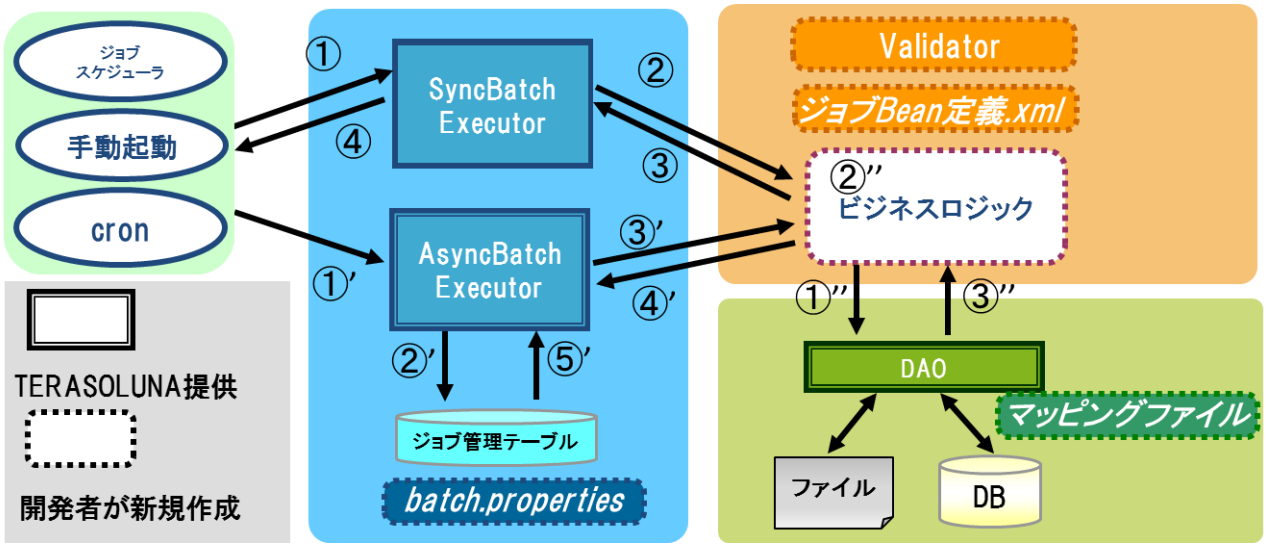
- BL-01 同期型ジョブ実行機能 **→BL-01参照**
  - SyncBatchExecutorを使用してジョブスケジューラなどからジョブを同期実行する機能を提供する。
- BL-02 非同期型ジョブ実行機能 **→BL-02参照**
  - AsyncBatchExecutorを利用してジョブ管理テーブルに登録されたジョブを非同期に実行する機能を提供する。
- BL-03 トランザクション管理機能 **→BL-03参照**
  - フレームワークがトランザクションを管理する方式を提供する。
  - ビジネスロジック内でトランザクションを管理できる方式を提供する。
- BL-04 例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、発生した例外をジョブ終了コードに変換する方式を提供する。
- BL-05 ビジネスロジック実行機能 **→BL-01, BL-03, BL-04参照**
  - 同期型ジョブ実行機能、非同期型ジョブ実行機能が実行するビジネスロジックのインタフェースを提供する。
  - 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**
  - ビジネスロジック内でデータベースにアクセスする方式を提供する。
- 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、文字列データ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**
  - 主にログに表示する文字列(メッセージリソース)を管理する機能を提供する。

Copyright © 2011-2016 NTT DATA CORPORATION.

- AL-041 入力データ取得機能 **→AL-041参照**
  - データベース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、データベース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。

Copyright © 2011-2016 NTT DATA CORPORATION.

◆ 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。

Copyright © 2011-2016 NTT DATA CORPORATION.

◆ 動作確認環境

- 対応JDK
  - Java SE7/8
- 対応データベース
  - Oracle 12c
  - PostgreSQL 9.3.x / 9.4.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	3.6.0
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	3.6.0
terasoluna-filedao	ファイルアクセス機能を提供する	3.6.0
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	3.6.0
terasoluna-commons	ユーティリティ機能など共通機能を提供する	3.6.0

Copyright © 2011-2016 NTT DATA CORPORATION.

● 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
args4j	2.32
aspectjweaver	1.8.7
classmate	1.1.0
commons-beanutils	1.9.2
commons-collections	3.2.2
commons-dbcp2	2.1.1
commons-jxpath	1.3
commons-lang3	3.3.2
commons-pool2	2.4.2
dozer	5.5.1
dozer-spring	5.5.1
hibernate-validator	5.2.2.Final
javax.el	3.0.0
javax.inject	1
jboss-logging	3.3.0.Final
jcl-over-slf4j	1.7.13
logback-classic	1.1.3
logback-core	1.1.3
mybatis	3.3.0
mybatis-spring	1.2.3
slf4j-api	1.7.13
spring-aop	4.2.4.RELEASE
spring-beans	4.2.4.RELEASE
spring-context	4.2.4.RELEASE
spring-core	4.2.4.RELEASE
spring-expression	4.2.4.RELEASE
spring-jdbc	4.2.4.RELEASE
spring-tx	4.2.4.RELEASE
validation-api	1.1.0.Final

Copyright © 2011-2016 NTT DATA CORPORATION.

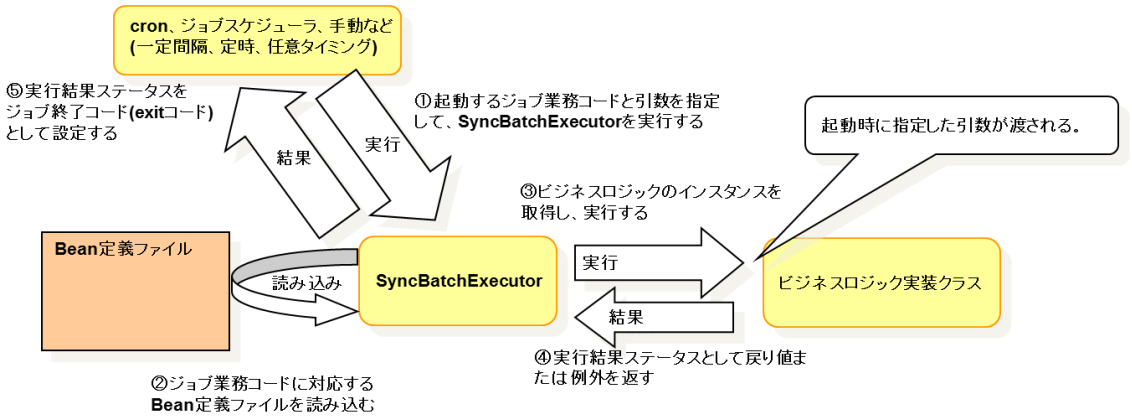
BL-01 同期型ジョブ実行機能

■ 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
- ジョブ業務コードを指定して特定のジョブを 1 件同期実行する
- 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、SyncBatchExecutor を実行する
cron やジョブスケジューラなどから引数を指定して SyncBatchExector を実行する。SyncBatchExecutor は与えられたジョブを同期実行するクラスである。ジョブの業務処理はビジネスロジック実装クラスに記述する。SyncBatchExecutor の引数と設定方法の一覧を以下に示す。

✧ SyncBatchExecutor の実行に必要な引数の一覧

実行時引数	環境変数 (※BL-01-1)	名称	必須	説明
第 1 引数	JOB_APP_CD	ジョブ業務コード	○	実行対象のジョブを一意に識別するための文字列
第 2～21 引数	JOB_ARG_NM 1～20	引数		ビジネスロジック実装クラスに与える引数

(※BL-01-1)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。

Copyright © 2011-2016 NTT DATA CORPORATION.

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
SyncBatchExecutor は、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み、アプリケーションコンテキストを生成する。たとえば、ジョブ業務コードを B000001 と設定した場合は、B000001.xml を読み込み、アプリケーションコンテキストを生成する。

③ ビジネスロジックのインスタンスを取得し、実行する
SyncBatchExecutor は、アプリケーションコンテキストから、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し、実行する。たとえば、ジョブ業務コードを B000001 と設定した場合は、B000001BLogic を実行する。

④ 実行結果ステータスとして戻り値または例外を返す
ビジネスロジックは、実行結果ステータスとして戻り値または例外を SyncBatchExecutor に返す。

⑤ 実行結果ステータスをジョブ終了コード(exit コード)として設定する
SyncBatchExecutor は、ビジネスロジックから返された戻り値をジョブ終了コード(exit コード)として起動元に返す。なお、例外が発生した時の終了コードについては、『BL-04 例外ハンドリング機能』を参照すること。

- 同期型ジョブ実行機能の設定

フレームワークは ApplicationResource.properties ファイルに設定されたプロパティファイルを読み込む。デフォルトではそのうちの batch.properties にフレームワークに関する設定が記述されている。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こと。なお、この値は後述する非同期型ジョブ実行機能と共通の設定となる。

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 定義ファイル(基本部)
beanDefinition.admin.dataSource	AdminDataSource.xml	管理用 Bean 定義ファイル(データソース部)
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 定義ファイルを配置するクラスパスはバッチ実行時に java の-D オプションで与えることもできる。

Copyright © 2011-2016 NTT DATA CORPORATION.

- 管理用 Bean 定義ファイル(データソース部)の読み込みについて

TERASOLUNA Batch 3.6.x より、同期型ジョブ実行機能の場合でも管理用 Bean 定義ファイル(データソース部)を読み込むようになった。これは、管理用 Bean 定義ファイル(基本部)に定義した非同期型ジョブ実行機能で使用するいくつかの Bean が管理用 Bean 定義ファイル(データソース部)の設定に依存するようになったためである。

これにより同期型ジョブ実行機能の実行に必要としない少量のデータソース関連の Bean が余分に生成されることになるが、この動作が性能上大きな問題を引き起こすことはなく、これまで提供してきた設定ファイルの構成を崩すほどのメリットがないと判断し、現在の構成にしている。

なお、デフォルト設定のまま使用しても、従来どおり同期型ジョブ実行機能の実行に管理用 Bean 定義ファイル(データソース部)の設定は従来どおり不要である。

この設定を変更する場合、ApplicationContextResolver の拡張が必要になる。詳細は、拡張ポイントの説明を参照すること。

Copyright © 2011-2016 NTT DATA CORPORATION.

■ 使用方法

◆ コーディングポイント

【コーディングポイントの構成】

- ジョブ起動シェルスクリプトの作成
  - 共通 CLASSPATH 定義シェルの定義
  - SyncBatchExecutor の実行

- ジョブ Bean 定義ファイルの設定
  - アノテーション設定の有効化
  - 共通コンテキストのインポート
  - データソース設定のインポート
  - コンポーネントスキャンの定義

- ビジネスロジックの実装
  - BLogic インタフェースを実装する
  - クラス名の宣言に@Component アノテーションを付与する
  - execute メソッドに業務処理を実装する

Copyright © 2011-2016 NTT DATA CORPORATION.

- ジョブ起動シェルスクリプトの作成

SyncBatchExecutor を実行するにはシェルスクリプトファイル(UNIX)またはバッチファイル(Windows)を実装する必要がある。本書では、Bourne Shell での設定例をもとに説明する。

➢ 共通 CLASSPATH 定義シェルの定義

SyncBatchExecutor の起動に必要になるライブラリはジョブ間で共通のものが多いため、共通 CLASSPATH 定義シェル(classpath.sh)を使用し、各ジョブ起動シェルスクリプト内で実行するようにするとよい。

✧ 共通 CLASSPATH 定義シェルの実装例

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

➢ SyncBatchExecutor の実行

共通 CLASSPATH 定義シェル(classpath.sh)を実行してから SyncBatchExecutor を実行する。実行に必要な引数は、実行時引数、または、環境変数のどちらかを使用して与える。なお、実運用にあたっては各種 Java オプション(-Xms や-Xmx など)を適切に使用すること。

ジョブ業務コード:B000001、引数:[2, 3, 4]の場合のジョブ起動スクリプトの実装例を以下に示す。

✧ 実行時引数を使用して SyncBatchExecutor を実行する場合

```
#!/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。
半角スペース区切りで設定する。

✧ 環境変数を使用して SyncBatchExecutor を実行する場合

```
#!/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し、実行時引数は設定しない。

Copyright © 2011-2016 NTT DATA CORPORATION.

- 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 SyncBatchExecutor が読み込めるように、「ジョブ業務コード」+「.xml」にする。ジョブ Bean 定義ファイルでは、最低限、以下の 3 つの設定が必要になる。

➢ アノテーション設定の有効化

Spring Framework のアノテーションによる設定を利用できるように<context:annotation-config />を設定する。

➢ 共通 Bean 定義ファイルの読み込み

ブランクプロジェクトのデフォルト提供ではジョブの実行に必要な 2 つの共通 Bean 定義ファイルを用意している。

- 共通コンテキスト

ビジネスロジックで利用する共通的な Bean を定義する Bean 定義ファイルを共通コンテキストと呼ぶ。ブランクプロジェクトのデフォルト提供では、beansDef ディレクトリ配下の commonContext.xml である。共通的な Bean を作成した場合のほか、ファイルアクセス機能で使用するファイル系 DAO や、例外ハンドリング機能で使用するデフォルト例外ハンドラなど、フレームワークの機能を使用する場合は共通コンテキストを読み込む必要がある。

- データソース設定

データソース関連の Bean を定義する Bean 定義ファイルをデータソース設定と呼ぶ。ブランクプロジェクトのデフォルト提供では、beansDef ディレクトリ配下の dataSource.xml である。ビジネスロジックでデータベースアクセス機能を使用する場合はデータソース設定を読み込む必要がある。

- 読み込み方法

TERASOLUNA Batch3.6.x 以降では、ApplicationContextResolver クラスの追加により、ジョブ Bean 定義ファイルの親となる Bean 定義ファイルを設定できるようになった。これにより、親となる Bean 定義ファイルに共通的な定義を行うことで、個々の Bean 定義ファイルの記述量を削減できるようになった。
ブランクプロジェクトのデフォルト設定では、共通コンテキスト、データソース設定の 2 つの Bean 定義ファイルを設定済みである。この設定により、ジョブ Bean 定義ファイル内でのこれら 2 つのファイルの読み込み設定が不要になる(<import>を設定する必要がない)。
なお、TERASOLUNA Batch3.5.x 以前では、ジョブ Bean 定義ファイルにインポートの記述が必要であった。

➢ コンポーネントスキャンの定義

実行するビジネスロジックは記述量削減のため、Spring Framework のコンポーネントスキャン機能を使用して定義するようになっている(Bean 定義をして

Copyright © 2011-2016 NTT DATA CORPORATION.

もよい)。<context:component-scan>タグを使用し、実行する業務ロジックが格納されたパッケージを base-package 属性に指定すると、ビジネスロジックの Bean 定義を省略できる。

以下に、ジョブ Bean 定義ファイルの実装例を示す。

✧ B000001.xml 実装例

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

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

jp.terasoluna.batch.sample.b000001 パッケージ配下にジョブ業務コード B000001 に対応するビジネスロジックがある場合の設定

Copyright © 2011-2016 NTT DATA CORPORATION.

- ビジネスロジックの実装

  - BLogic インタフェースを実装する

    SyncBatchExecutor から呼び出すビジネスロジックは、BLogic インタフェースに定義されている execute メソッドを呼び出す決まりとなっている。この execute メソッドに業務処理を実装する。

    なお、トランザクション管理機能を使用してトランザクションをフレームワークで管理する場合は、AbstractTransactionBLogic クラスを継承して作成することになる。詳細は後述するトランザクション管理機能の説明を参照すること。

  - クラス名の宣言に@Componet アノテーションを付与する

    ジョブ Bean 定義ファイルに定義したコンポーネントスキャンの対象とするためには、@Component アノテーションを付与する必要がある。

  - execute メソッドに業務処理を実装する

    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。戻り値は int 型の整数であり、起動元に返却されるジョブ終了コードとなる。ビジネスロジックからは、@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(コンポーネントスキャンした)Bean をフィールドにインジェクションして使用できる。

    ✧ B000001BLogic 実装例

```
@Component
public class B000001BLogic implements BLogic {

  @Inject
  B000001Dao b000001Dao = 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@Injcet 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@Named と併用して利用すること。

たとえば、同じ型で 2 つの Bean(「b000001Dao_1」と「b000001Dao_2」とする)が定義されていた場合、「b000001Dao_1」をフィールドにインジェクションしたい時は、以下のようにする。

```
@Inject
@Named("b000001Dao_1")
B000001Dao b000001Dao_1 = null
```

なお、ByName でインジェクションする方法は、@Inject アノテーションと@Named アノテーションを併用する以外にも、@Resource アノテーションを使用する方法がある。

BLogic の戻り値がジョブ終了コードとして返却される。

Copyright © 2011-2016 NTT DATA CORPORATION.

■ リファレンス

◆ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.batch.executor.SyncBatchExecutor	与えられたジョブを同期実行する同期型ジョブ実行機能のエントリポイントとなるクラス。
2	jp.terasoluna.fw.batch.executor.ApplicationContextResolver	アプリケーションコンテキストを解決するためのクラス。
3	jp.terasoluna.fw.batch.executor.ApplicationContextResolverImpl	アプリケーションコンテキストを解決するためにフレームワークが提供するデフォルトの実装クラス。
4	jp.terasoluna.fw.batch.executor.controller.JobOperator	ジョブの実行を管理するためのインタフェース。
5	jp.terasoluna.fw.batch.executor.controller.SyncJobOperatorImpl	同期型ジョブの実行を管理するためにフレームワークが提供するデフォルトの実装クラス。
6	jp.terasoluna.fw.batch.executor.vo.BatchJobData	ジョブの実行に必要なパラメータを保持するためのクラス。
7	jp.terasoluna.fw.batch.blogic.BLogicResolver	ビジネスロジックを解決するためのインタフェース。
8	jp.terasoluna.fw.batch.blogic.BLogicResolverImpl	ビジネスロジックを解決するためにフレームワークが提供するデフォルトの実装クラス。
9	jp.terasoluna.fw.batch.blogic.vo.BLogicParamConverter	ジョブの実行に必要なパラメータをビジネスロジック実行時の入力パラメータに変換するためのインタフェース。
10	jp.terasoluna.fw.batch.blogic.vo.BLogicParamConverterImpl	ジョブの実行に必要なパラメータをビジネスロジック実行時の入力パラメータに変換するためにフレームワークが提供するデフォルトの実装クラス。
11	jp.terasoluna.fw.batch.blogic.vo.BLogicParam	ビジネスロジック実行時の入力パラメータを保持するクラス。
12	jp.terasoluna.fw.batch.executor.BLogicExecutor	ビジネスロジックを実行し、実行結果を戻り値として取得するためのインタフェース。
13	jp.terasoluna.fw.batch.executor.BLogicExecutorImpl	ビジネスロジックを実行し、実行結果を戻り値として取得するためにフレームワークが提供するデフォルトの実装クラス。

Copyright © 2011-2016 NTT DATA CORPORATION.

14	jp.terasoluna.fw.batch.executor.vo.BLogicResult	ビジネスロジックの実行結果を保持するクラス。

◆ 拡張ポイント

- 管理用 Bean 定義ファイルの読み込みについて

TERASOLUNA Batch 3.6.x 以降では、ApplicationContextResolver クラスの追加により、管理用 Bean 定義ファイル(基本部)だけでなく、同期型ジョブ実行機能の実行に本来必要のない管理用 Bean 定義ファイル(データソース部)を読み込んでいる(その理由は、前述の「管理用 Bean 定義ファイル(データソース部)の読み込みについて」を参照)。

TERASOLUNA Batch 3.5.x までのように、管理用 Bean 定義ファイル(基本部)だけを読み込みたい場合は、以下のような方法がある。

- 非同期型ジョブ実行機能を使用しない場合
  管理用 Bean 定義ファイル(データソース部)の Bean 定義を空定義(<beans>のみ)とし、管理用 Bean 定義ファイル(基本部)から、"async"ではじまる非同期型ジョブ実行機能に関連する Bean 定義を削除する。

- 非同期型ジョブ実行機能を使用する場合
  現在の設定ファイルの構成を管理用 Bean 定義ファイルを同期型ジョブ実行機能の実行に必要な Bean のみを定義したファイル、非同期型用ジョブ実行機能の実行に必要な Bean を追加で定義したファイルに分ける。なお、この方法は SyncBatchExecutor や AsyncBatchExecutor(後述)から呼び出される ApplicationContextResolverImpl を拡張する必要がある。

■ 関連機能

- 『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)

Copyright © 2011-2016 NTT DATA CORPORATION.

■ 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でき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。

✧ ジョブ業務コード B00001 に対して SampleBLogic を使用する例

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

@JobComponent にジョブ ID を付与して使用する。

● 注意点

@JobComponent アノテーションの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。

✧ 機能の有効化 (batchapps/batch.properties)

```
enableJobComponentAnnotation=true
```

Copyright © 2011-2016 NTT DATA CORPORATION.

◆ 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。

- SyncBatchExecutor 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SyncBatchExecutor 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、SyncBatchExecutor を起動し、ジョブを再実行する。なお、フレームワークには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ジョブのリカバリ設計(異常終了した際の途中データを削除してから再実行するなど)をしておくこと。

◆ ApplicationContextResolver の設定について

TERASOLUNA Batch 3.6.x 以降で提供する ApplicationContextResolver では、ジョブ Bean 定義ファイルの親となる Bean 定義ファイルを設定することができる。

✧ 設定例 (beansAdminDef/AdminContext.xml)

```
<!-- ジョブ Bean 定義ファイルの解決クラス -->
<bean id="blogicApplicationContextResolver"
      class="jp.terasoluna.fw.batch.executor.ApplicationContextResolverImpl">
      <property name="commonContextClassPath"
            value="classpath:beansDef/commonContext.xml,
                  classpath:beansDef/dataSource.xml" />
</bean>
```

カンマ区切りで複数指定可能

上記のように設定した場合、以下の挙動となる。

1. フレームワークの起動時に管理用 Bean 定義ファイルにもとづくアプリケーションコンテキストを生成する。
2. 1.の際に、commonContextClassPath プロパティで指定した Bean 定義ファイルを生成する。
3. その後、ジョブ Bean 定義ファイルにもとづくアプリケーションコンテキストを生成する際に、2.で生成したアプリケーションコンテキストを親として設定する。

Copyright © 2011-2016 NTT DATA CORPORATION.

なお、commonContextPath プロパティに Bean 定義ファイルを設定していない場合は 2.は行わず、3.でジョブ Bean 定義ファイルにもとづくアプリケーションアプリケーションコンテキストの生成する際に親を設定しない。

TERASOLUNA Batch 3.5.x までは、<import>を使用し、ジョブ Bean 定義ファイルに共通コンテキストやデータソース設定の Bean 定義をインポートしていた。この場合、共通コンテキストやデータソース設定で定義している Bean をジョブ Bean 定義ファイル内に定義したことと同じである。つまり、ジョブ Bean 定義ファイルにもとづくアプリケーションコンテキストを生成するたびに共通コンテキストやデータソース設定で定義した Bean を生成していた。

◆ TERASOLUNA Batch 3.5.x 以前のジョブ Bean 定義ファイルをそのまま使用する際の注意

TERASOLUNA Batch 3.5.x 以前のジョブ Bean 定義ファイルをそのまま使用する場合、ApplicationContextResolver の Bean 定義にデフォルトで設定されている共通コンテキストとデータソース設定の指定を削除すること。

ApplicationContextResolver の Bean 定義がデフォルト設定のまま TERASOLUNA Batch 3.5.x 以前のジョブ Bean 定義ファイルをそのまま使用した際は、共通コンテキストとデータソース設定で定義した Bean が以下のタイミングで二重に生成される。

- 管理用 Bean 定義ファイルにもとづくアプリケーションコンテキストの生成時
- ジョブ Bean 定義ファイルにもとづくアプリケーションコンテキストの生成時

Bean の生成数が多くなると、それだけ初期処理に時間がかかるため、必要な分だけ Bean を生成するように注意すること。

Copyright © 2011-2016 NTT DATA CORPORATION.

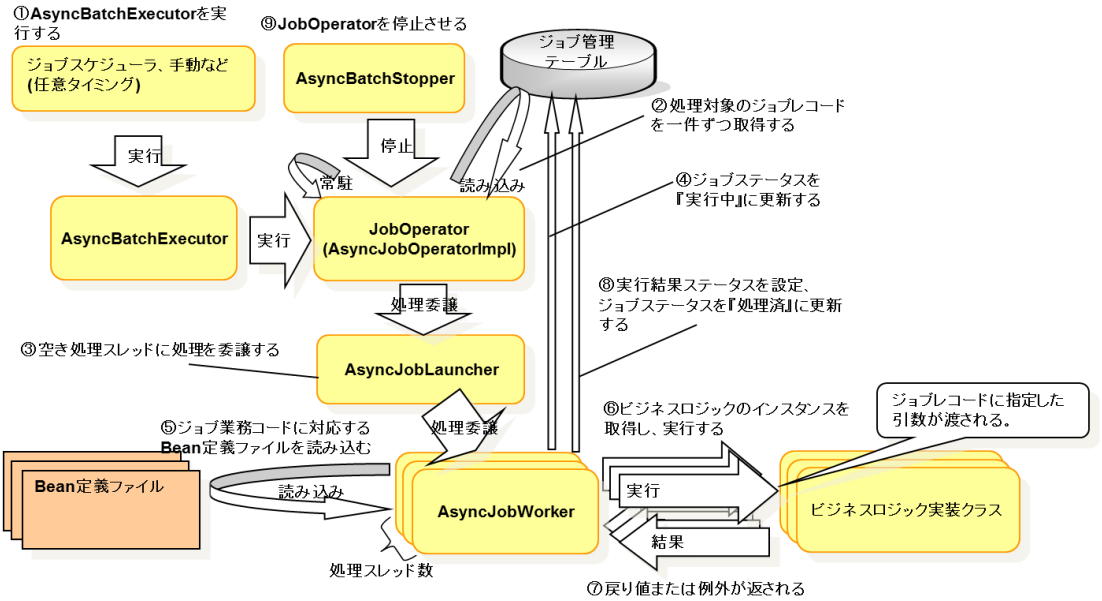
BL-02 非同期型ジョブ実行機能

■ 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
- データベースに作成したジョブ管理テーブルを定期的に監視し、ジョブ管理テーブルに登録されている複数のジョブをマルチスレッドで多重実行する
- ジョブの処理スレッドはメインスレッドとは異なる

◆ 概念図



◆ 解説

- 非同期型ジョブの起動から終了までの流れ

① AsyncBatchExecutor を実行する

cron やジョブスケジューラなどから、AsyncBatchExecutor を実行する。AsyncBatchExecutor は JobOperator を起動する。JobOperator は、後述の⑨で停止するまでジョブ管理テーブルの監視を続ける。

Copyright © 2011-2016 NTT DATA CORPORATION.

② 処理対象のジョブレコードを一件ずつ取得する
JobOperator はジョブ管理テーブルから処理対象(ジョブステータスが『未実行』のジョブ)を 1 件取得し、AsyncJobLauncher にジョブの実行を委譲する。

③ 空き処理スレッドに処理を委譲する
AsyncJobLauncher のデフォルト実装クラスである AsyncJobLauncherImpl は、Spring Framework の ThreadPoolTaskExecutor を使用して JobOperator から委譲されたジョブを非同期に実行する。実際のジョブの実行は AsyncJobWorker に委譲する。AsyncJobWorker が動作するスレッドの数(多重度)は管理用 Bean 定義ファイル(基本部)の非同期型ジョブ実行用スレッドプールタスクエグゼキュータの設定で調整する。

✧ Bean 定義の例(beansAdminDef/AdminContext.xml)

```
<!-- 非同期型ジョブ実行用スレッドプールタスクエグゼキュータ -->
<task:executor id="batchTaskExecutor"
                pool-size="10-10"
                queue-capacity="10" />
```

✧ 設定値の説明

プロパティ	説明
pool-size	コアスレッド数とスレッドの最大許容数を、n-m の形で設定する。たとえば、コアスレッド数が 5、スレッドの最大許容数が 10 の場合は、5-10 と設定する。
queue-capacity	内部キューの容量を指定する。 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。 queue-capacity はスレッドの最大許容数以上の値を設定しておくことを推奨する。スレッドの最大許容数より小さい値を設定すると、TaskRejectExeception が発生し、ジョブが実行されない可能性がある。

④ ジョブのステータスを『実行中』に更新する
AsyncJobWorker はジョブの実行を開始するため、ジョブのステータスを『未実行』から『実行中』に更新する。このとき、既に他のスレッドでこのジョブの実行が開始されている場合は、ジョブの実行をスキップする。

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の「job_app_cd」カラムから取得した実行対象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み、アプリケーションコンテキストを生成する。

Copyright © 2011-2016 NTT DATA CORPORATION.

⑥ ビジネスロジックのインスタンスを取得し、実行する。
AsyncJobWorker は、アプリケーションコンテキストから、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し、実行する。戻り値もしくは例外が返される

⑦ 戻り値または例外が返される
ビジネスロジックは、実行の結果を戻り値または例外として AsyncJobWorker に返す。

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。
AsyncJobWorker は、⑦の結果をジョブ管理テーブルの「blogic_app_status」カラムに登録し、ジョブステータスを『処理済み』に更新する。

⑨ JobOperator を停止させる
非同期型ジョブ実行機能の終了には JobOperator の停止が必要になる。JobOperator が終了するタイミングは、AsyncBatchStopper による判断、または、異常終了がある。詳細は、後述の「非同期型ジョブ実行機能の終了」を参照すること。

- ジョブ管理テーブル

ジョブ管理テーブルのデフォルト定義を以下に示す。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。

✧ ジョブ管理テーブルのデフォルト定義

	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与える引数
	…	…		
22	引数 20	job_arg_nm20		ビジネスロジックに与える引数
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

Copyright © 2011-2016 NTT DATA CORPORATION.

- 非同期型ジョブ実行機能の設定

フレームワークは ApplicationResource.properties ファイルに設定されたプロパティファイルを読み込む。デフォルトではそのうちの batch.properties にフレームワークに関する設定が記述されている。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こと。

✧ 設定値の説明

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 定義ファイル(基本部)
beanDefinition.admin.dataSource	AdminDataSource.xml	管理用 Bean 定義ファイル(データソース部)
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 定義ファイルを配置するクラスパスはバッチ実行時に java の -D オプションで与えることもできる。
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.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)

Copyright © 2011-2016 NTT DATA CORPORATION.

- 非同期型ジョブ実行機能の終了
  - AsyncBatchStopper による判断
    - 終了ファイルの配置

      終了ファイル(プロパティ「executor.endMonitoringFile」に設定したファイル。空ファイルでよい)を配置して非同期型ジョブ実行機能を終了する。
      非同期型ジョブ実行機能の実行後、AsyncBatchStopper は定期的に終了ファイルが存在しているかどうかをチェックする。その際に終了ファイルが存在していた場合は、終了と判定する。終了と判定された後は、現在実行中のジョブが終了するのを待って、非同期型ジョブ実行機能を終了する(新規ジョブの実行は行わない)。
      たとえば、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 というファイルを配置すれば、非同期型ジョブ実行機能は終了する。
  - 異常終了
    - 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

      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

      DB サーバが途中でシャットダウンした場合、デフォルトでは非同期型ジョブ実行機能は定期的なジョブ管理テーブルの監視ができなくなるため、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。

Copyright © 2011-2016 NTT DATA CORPORATION.

- 非同期型ジョブ実行機能のリトライ機能

プロパティの値を変更することで、DB サーバのシャットダウンなどによりジョブ管理テーブルとの通信が切断された際に DB サーバへの接続をリトライすることができる。

JobControlFinder や JobStatusChanger が AdminConnectionRetryInterceptor の実行対象となるように AOP の定義を追加することで、本機能を使用できる。

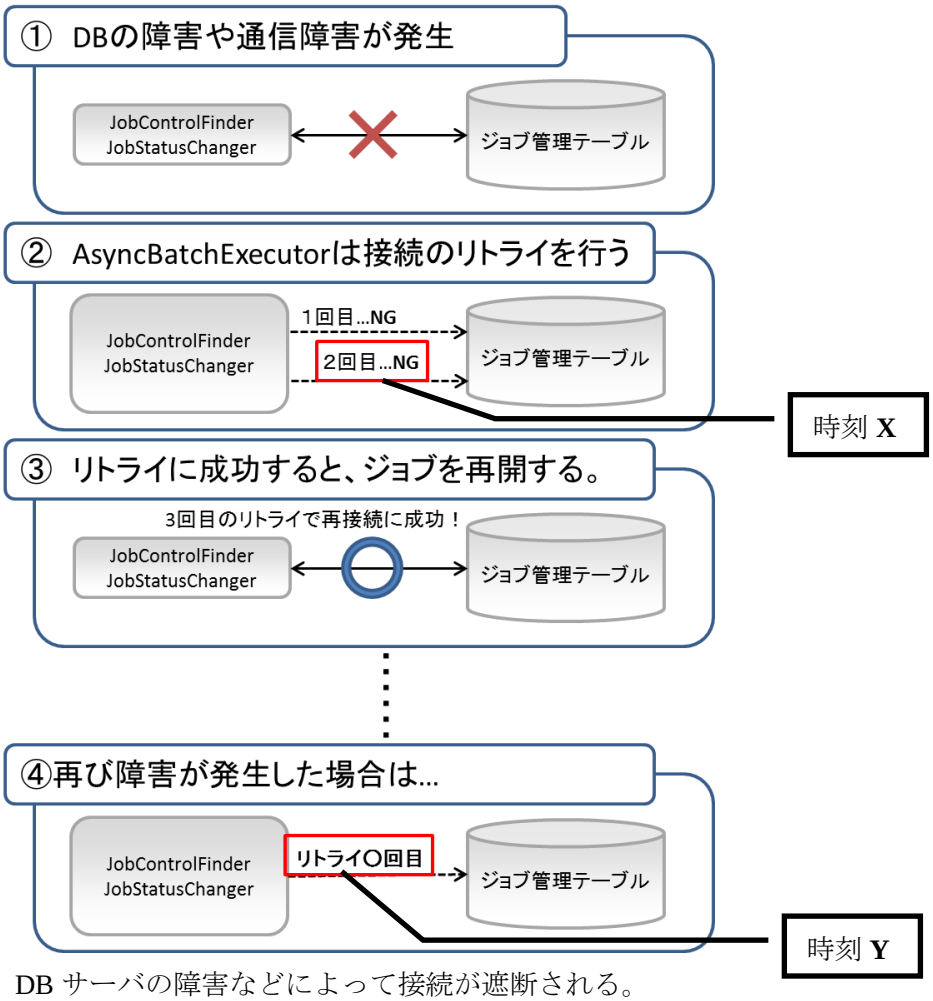
✧ Bean 定義の例(beansAdminDef/AdminContext.xml)

```
<!-- データベース異常時のリトライ機能(非同期型ジョブ実行機能用) AOP 定義 -->
<bean id="adminConnectionRetryInterceptor"
    class="jp.terasoluna.fw.batch.executor.AdminConnectionRetryInterceptor" />
<aop:config>
    <aop:pointcut id="adminConnectionRetryPointcut" expression="
        execution(*jp.terasoluna.fw.batch.executor.repository.JobStatusChanger.*(..))
        || execution(* jp.terasoluna.fw.batch.executor.repository.JobControlFinder.*(..))" />
    <aop:advisor advice-ref="adminConnectionRetryInterceptor"
                pointcut-ref="adminConnectionRetryPointcut" />
</aop:config>
```

リトライ機能のイメージを説明するために、次のページで以下の設定をした場合のリトライの流れを示す。

batchTaskExecutor.dbAbnormalRetryMax=3
batchTaskExecutor.dbAbnormalRetryInterval=20000(デフォルト値)
batchTaskExecutor.dbAbnormalRetryReset=600000(デフォルト値)

Copyright © 2011-2016 NTT DATA CORPORATION.



① DB サーバの障害などによって接続が遮断される。

② AdminConnectionRetryInterceptor は 20000 ミリ秒間隔(※BL-02-1)で接続のリトライを試みる(2 回のリトライで繋がったと仮定する)。
(※BL-02-1)batchTaskExecutor.dbAbnormalRetryInterval に設定された値
##この時 2 回目のリトライで NG となった時刻を「X」とする##

③ 3 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た場合は、今回のリトライ時刻から前回のリトライ時刻 X を差し引いた値が 600000 ミリ秒(※BL-02-2)を上回っていれば AdminConnectionRetryInterceptor はリトライ回数をリセットし、再びリトライを試みる。下回っていた場合はリトライ回数が 3 を超えるためリトライを試みず、例外をスローする。
(※BL-02-2)batchTaskExecutor.dbAbnormalRetryReset に設定された値

Copyright © 2011-2016 NTT DATA CORPORATION.

- 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。

Copyright © 2011-2016 NTT DATA CORPORATION.

■ 使用方法

◆ コーディングポイント

【コーディングポイントの構成】

- ジョブ起動シェルスクリプトの作成
  - 共通 CLASSPATH 定義シェルの定義
  - SyncBatchExecutor の実行
- データベースへの接続設定
  - 接続情報の編集
  - システム利用 DAO の設定
- ジョブ 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)

Copyright © 2011-2016 NTT DATA CORPORATION.

- ジョブ起動シェルスクリプトの作成

AsyncBatchExecutor を実行するにはシェルスクリプトファイル(UNIX)またはバッチファイル(Windows)を実装する必要がある。本書では、Bourne Shell での設定例をもとに説明する。

➢ 共通 CLASSPATH 定義シェルの定義

AsyncBatchExecutor の起動に必要になるライブラリは同期型ジョブ実行機能と共通のため、同期型ジョブ実行機能で使用する共通 CLASSPATH 定義シェル(classpath.sh)を使用し、各ジョブ起動シェルスクリプト内で実行するようにするとよい。

✧ 共通 CLASSPATH 定義シェルの実装例

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

➢ AsyncBatchExecutor の起動

共通 CLASSPATH 定義シェル(classpath.sh)を実行してから AsyncBatchExecutor を起動する。なお、実運用にあたっては各種 Java オプション(-Xms や-Xmx など)を適切に使用すること。

✧ AsyncBatchExecutor の起動例

```
#!/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
```

- データベースへの接続設定

➢ 接続情報の編集

ジョブ管理テーブルにアクセスするため、データベースへの接続設定が必要となる。デフォルトでは mybatisAdmin/jdbc.properties に記載する。

✧ データベースの接続例 (mybatisAdmin/jdbc.properties)

```
jdbc.driver=org.postgresql.Driver
jdbc.url=jdbc:postgresql://127.0.0.1:5432/postgres
jdbc.username=postgres
jdbc.password=postgres
```

➢ システム利用 DAO の設定

使用するデータベースの種類は PostgreSQL がデフォルトになっているが、Oracle や他の設定 DBMS に変更する場合は、管理 Bean 定義ファイル(データソース部)のシステム利用 DAO の設定や SQL も変更すること。

Copyright © 2011-2016 NTT DATA CORPORATION.

✧ データベースの切り替え (beansDef/AdminDataSource.xml)

```
<!-- システム利用 DAO 定義(Oracle)
<bean id="systemDao" class="org.mybatis.spring.mapper.MapperFactoryBean">
    <property name="mapperInterface"
        value="jp.terasoluna.fw.batch.executor.dao.SystemOracleDao" />
    <property name="sqlSessionFactory" ref="sysSqlSessionFactory" />
</bean>
-->

<!-- システム利用 DAO 定義(PostgreSQL) -->
<bean id="systemDao" class="org.mybatis.spring.mapper.MapperFactoryBean">
    <property name="mapperInterface"
        value="jp.terasoluna.fw.batch.executor.dao.SystemPostgreSQLDao" />
    <property name="sqlSessionFactory" ref="sysSqlSessionFactory" />
</bean>
```

使用する定義を有効にすること。

なお、Oracle、PostgreSQL 以外の DBMS を使用する場合や、ジョブ管理テーブルをカスタマイズした場合は、システム利用 DAO が使用するジョブ管理テーブルにアクセスするための SQL を見直す必要がある。データベースアクセスの仕組みの詳細は、『BL-06 データベースアクセス機能』を参照すること。

✧ SQL の例 (jp/terasoluna/fw/batch/executor/dao/SystemPostgreSQLDao.xml)

```
<mapper
    namespace="jp.terasoluna.fw.batch.executor.dao.SystemPostgreSQLDao">
    <!-- ジョブリスト取得 -->
    <select id="selectJobList" parameterType="BatchJobListParam"
        resultType="BatchJobListResult">
        SELECT
            A.JOB_SEQ_ID AS jobSequenceId
        FROM
            JOB_CONTROL A
… (省略) …
        ORDER BY
            A.CUR_APP_STATUS DESC,
            A.JOB_SEQ_ID
    </select>
</mapper>
```

ジョブ管理テーブルにアクセスするための SQL 定義(一部)

- ジョブ Bean 定義ファイルの設定

『BL-01 同期型ジョブ実行機能』と同様である。

- ビジネスロジックの実装

『BL-01 同期型ジョブ実行機能』と同様である。

Copyright © 2011-2016 NTT DATA CORPORATION.

■ リファレンス

◆ 構成クラス

同期型ジョブ実行機能と共通で使用するクラスの説明は割愛する。

	クラス名	概要
1	jp.terasoluna.fw.batch.executor.AsyncBatchExecutor	ジョブ管理テーブルに登録されたジョブを非同期実行する非同期型ジョブ実行機能のエントリポイントとなるクラス。
2	jp.terasoluna.fw.batch.executor.CacheableApplicationContextResolverImpl	アプリケーションコンテキストを解決するためにフレームワークが提供する業務コンテキストのキャッシュ機能つき実装クラス。
3	jp.terasoluna.fw.batch.executor.controller.AsyncJobOperatorImpl	非同期型ジョブの実行を管理するためにフレームワークが提供するデフォルトの実装クラス。
4	jp.terasoluna.fw.batch.executor.controller.AsyncJobLauncher	非同期型ジョブを起動するためのインタフェース。
5	jp.terasoluna.fw.batch.executor.controller.AsyncJobLauncherImpl	非同期型ジョブを実行するためにフレームワークが提供するデフォルトの実装クラス。ThreadPoolTaskExecutor を使用して非同期型ジョブを実行する。
6	jp.terasoluna.fw.batch.executor.AsyncJobWorker	非同期型ジョブをワーカースレッド内で実行するためのインタフェース。
7	jp.terasoluna.fw.batch.executor.AsyncJobWorkerImpl	非同期型ジョブをワーカースレッド内で実行するためにフレームワークが提供するデフォルトの実装クラス。ジョブ管理テーブルを使用して指定されたジョブシーケンスコードに紐づくジョブを実行する。
8	jp.terasoluna.fw.batch.executor.controller.AsyncBatchStopper	非同期型ジョブ実行機能の終了判定を行うためのインタフェース。
9	jp.terasoluna.fw.batch.executor.controller.EndFileStopper	非同期型ジョブ実行機能の終了判定を行うためにフレームワークが提供するデフォルトの実装クラス。終了ファイルを使用して非同期型ジョブ実行機能を停止する。
10	jp.terasoluna.fw.batch.executor.dao.SystemDao	フレームワークによるデータベースアクセス時に使用される DAO インタフェース。
11	jp.terasoluna.fw.batch.executor.dao.SystemOracleDao	フレームワークによるデータベースアクセス時に使用される DAO インタフェース(Oracle 用)

Copyright © 2011-2016 NTT DATA CORPORATION.

12	jp.terasoluna.fw.batch.executor.dao.SystemPostgreSQLDao	フレームワークによるデータベースアクセス時に使用される DAO インタフェース(PostgreSQL 用)
13	jp.terasoluna.fw.batch.executor.repository.JobControlFinder	ジョブパラメータを解決するためのインタフェース。
14	jp.terasoluna.fw.batch.executor.repository.JobControlFinderImpl	ジョブパラメータを解決するためにフレームワークが提供するデフォルトの実装クラス。ジョブ管理テーブルを使用してジョブパラメータを解決する。
15	jp.terasoluna.fw.batch.executor.vo.BatchJobListParam	ジョブ管理テーブルから実行対象のジョブのレコードを取得するための入力パラメータクラス。
16	jp.terasoluna.fw.batch.executor.vo.BatchJobListResult	ジョブ管理テーブルから実行対象のジョブのレコードを取得した結果を保持するためのクラス。
17	jp.terasoluna.fw.batch.executor.vo.BatchJobManagementParam	ジョブ管理テーブルからレコードを 1 件取得するための入力パラメータクラス。
18	jp.terasoluna.fw.batch.executor.vo.BatchJobManagementUpdateParam	ジョブ管理テーブルのレコードを更新するための入力パラメータを保持するクラス。
19	jp.terasoluna.fw.batch.executor.repository.JobStatusChanger	ジョブの実行ステータスを更新するためのインタフェース。
20	jp.terasoluna.fw.batch.executor.repository.JobStatusChangerImpl	ジョブの実行ステータスを更新するためにフレームワークが提供するデフォルトの実装クラス。ジョブ管理テーブルのジョブステータスカラムを更新する。
21	jp.terasoluna.fw.batch.executor.AdminConnectionRetryInterceptor	フレームワークによるデータベースアクセス時に異常があった際にリトライさせるためのインターセプター。

Copyright © 2011-2016 NTT DATA CORPORATION.

◆ 拡張ポイント

- ジョブ管理テーブルのカスタマイズによる拡張

ジョブ管理テーブルは自由にカスタマイズすることができる。たとえば、カラムを追加と、システム利用 DAO することにより、以下のような拡張ができる。

- グループ ID カラムを追加し、処理対象のジョブをグルーピングする。
- 優先度カラムを追加し、優先度の高いジョブを優先して処理対象とするように制御する。

- JobControlFinder の拡張

ジョブ管理テーブルから処理対象のジョブを取得する JobControlFinder は AsyncBatchExecutor に与えられた引数を使用することができる。たとえば、デフォルト実装では取得対象のジョブを限定していないが、JobControlFinder の実装を差し替えることにより、以下のような拡張ができる。

- AsyncBatchExecutor の第 1 引数を使用して処理対象のジョブ業務コードを限定する

■ 関連機能

- 『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』
- 『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)

Copyright © 2011-2016 NTT DATA CORPORATION.

■ 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』と同様である。

◆ 同じジョブを短時間に連続して実行する場合について

非同期型ジョブ実行機能を使用して同じジョブを短時間に連続して実行する場合は、CacheableApplicationContextResolverImpl を使用すると、性能の向上が見込める。

使用に際しては、Spring Cache Abstraction の設定が追加で必要になる。

✧ Spring Cache Abstraction の設定(Bean 定義ファイル)

```
<cache:annotation-driven proxy-target-class="true" />
<bean id="cacheManager"
        class="org.springframework.cache.support.SimpleCacheManager">
        <property name="caches">
          <set>
            <bean class="
                org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean">
                  <!-- 業務コンテキストのキャッシュ名は businessContext 固定 -->
                  <property name="name" value="businessContext" />
            </bean>
          </set>
      </property>
</bean>
<bean
    id="blogicApplicationContextResolver"
    class="jp.terasoluna.fw.batch.executor.CacheableApplicationContextResolverImpl">
    <!-- 共通コンテキストを業務コンテキストの親とする場合、
         commonContextClassPath で Bean 定義ファイルのクラスパスを記述する。
         (複数指定時はカンマ区切り) -->
        <property
            name="commonContextClassPath"
            value="beansDef/commonContext.xml,beansDef/dataSource.xml" />
        <!-- cacheManager の setter-injection -->
        <property name="cacheManager" ref="cacheManager"/>
</bean>
```

Copyright © 2011-2016 NTT DATA CORPORATION.

◆ ApplicationContextResolver の設定について

『BL-01 同期型ジョブ実行機能』と同様である。

なお、ApplicationContextResolver の commonContextPath プロパティに設定した Bean 定義ファイルは AsyncBatchExecutor の起動時に 1 回だけ生成される。TERASOLUNA Batch 3.5.x 以前では非同期型ジョブの実行のたびに生成されていたため、効率的となった。

◆ 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。

- AsyncBatchExecutor の異常を検知する

ジョブスケジューラ等で AsyncBatchExecutor プロセスの死活監視を行い、AsyncBatchExecutor 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。

- AsyncBatchExecutor 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 AsyncBatchExecutor 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、AsyncBatchExecutor 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。AsyncBatchExecutor の起動後、再実行対象となる。

なお、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。

Copyright © 2011-2016 NTT DATA CORPORATION.

BL-03 トランザクション管理機能

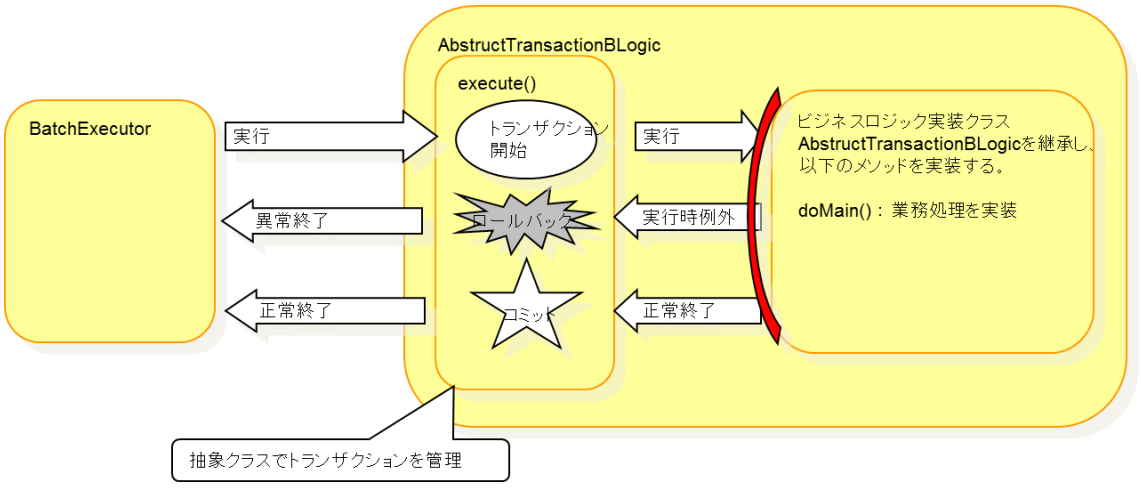
■ 概要

◆ 機能概要

- フレームワークで以下の2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。

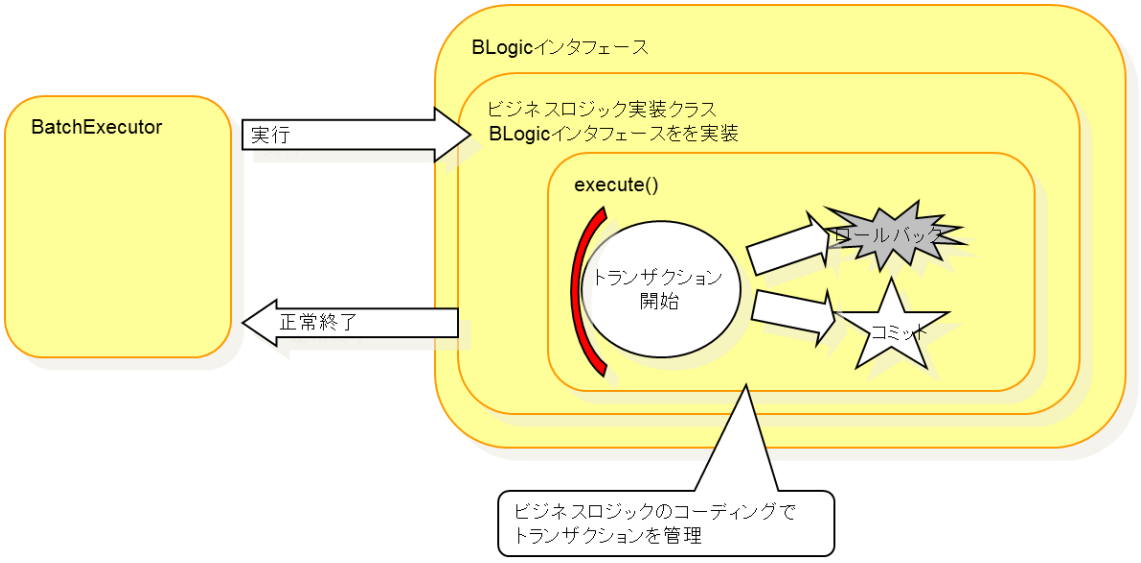
◆ 概念図

- 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



Copyright © 2011-2016 NTT DATA CORPORATION.

- 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

- 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。ビジネスロジック実行中に実行時例外が発生した場合は、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てビジネスロジックを実装する

✧ 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、ビジネスロジック中で任意に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。

Copyright © 2011-2016 NTT DATA CORPORATION.

■ 使用方法

◆ コーディングポイント

【コーディングポイントの構成】

- Bean 定義ファイルの設定
  - データソースの設定
  - トランザクションマネージャの設定
- 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
  - 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

Copyright © 2011-2016 NTT DATA CORPORATION.

- Bean 定義ファイルの設定

ジョブの起動方法やトランザクションモデルに関わらず、Bean 定義ファイルの設定が必要になる。

➢ データソースの設定

トランザクション管理機能で管理する対象のデータソースの設定を行う。詳細は『BL-06 データベースアクセス機能』を参照すること。

➢ トランザクションマネージャの設定

トランザクションは Spring Framework が提供する PlatformTransactionManager インタフェースを実装するトランザクションマネージャを使用して管理する。

そのなかでも、単一のデータソース(※BL-03-1)に対してトランザクションを管理するトランザクションマネージャの DataSourceTransactionManager を使用する。

(※BL-03-1)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。

✧ 設定例(beansDef/dataSource.xml)

```
<!--データソースの設定 -->
<bean id="dataSource" class="……">……</bean>

<!--トランザクションマネージャの設定 -->
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

管理対象のデータソースを指定する。

Copyright © 2011-2016 NTT DATA CORPORATION.

- 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 を継承したクラスを作成し、doMain メソッドをオーバライドして、業務処理を実装する。

    doMain メソッドがフレームワークから呼び出される前にトランザクションが開始され、doMain メソッドから戻り値が返った(正常終了した)後で、トランザクションがコミットされる。

    トランサクションをロールバックしたい場合は、例外をスローする。ただし、ビジネスロジックからは非検査例外しかスローできないことに留意すること。共通的な非検査例外として、フレームワークは BatchException を提供している。

    ✧ ビジネスロジックの実装例

```
@Component
public class B000001BLogic extends AbstractTransactionBLogic {

    @Override
    public int doMain(BLogicParam param) {
      try {
          //業務処理
          … (省略) …
      } 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。

例外が発生した場合はロールバックされる。

例外が発生しなかった場合はコミットされる。

Copyright © 2011-2016 NTT DATA CORPORATION.

- 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 インタフェースの実装クラスを作成し、execute メソッドに業務処理を実装する。

フレームワークではトランザクションの管理を行わないので、execute メソッド内で Bean 定義ファイルで設定した DataSourceTransactionManager とフレームワーク提供の BatchUtil を使用してトランザクションの開始・終了やコミット・ロールバックを行う。

✧ ビジネスロジックの実装例(一括コミット)

```
@Component
public class B000001BLogic implements BLogic {

    @Inject
    PlatformTransactionManager transactionManager = null;

    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
            … (省略) …

            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);
        }
    }
}
```

トランザクションマネージャはビジネスロジックのフィールドに定義してインジェクションする。

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。

エラーとなった場合のロールバック

正常終了の場合はコミット

例外が発生した場合など、コミットもロールバックもされていない場合は、ロールバックしてトランザクションを終了させる。

Copyright © 2011-2016 NTT DATA CORPORATION.

コミットを分割する場合は、以下のようにコミットした後で、トランザクションを再度開始すること。また、TransactionStatus は必ず正しいものを設定すること。

✧ ビジネスロジックの実装例(分割コミット)

```
@Component
public class B000002BLogic implements BLogic {

    @Inject
    PlatformTransactionManager transactionManager = null;


    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);
        }
    }
}
```

必ず正しい TransactionStatus を設定すること。設定せずにトランザクションを操作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・ロールバックされない

Copyright © 2011-2016 NTT DATA CORPORATION.

■ リファレンス

◆ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.batch.blogic.BLogic	同期型ジョブ実行機能/非同期型ジョブ実行機能から実行されるビジネスロジックを規定するインタフェース。 トランザクション管理を行わない場合や、ビジネスロジックで任意にトランザクションを管理したい場合はBLogic インタフェースを使用してビジネスロジックを実装する。
2	jp.terasoluna.fw.batch.blogic.AbstractTransactionBLogic	業務処理の前後にトランザクション管理を行う処理を実装した BLogic インタフェースの抽象クラス。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はAbstractTransactionBLogic クラスを使用してビジネスロジックを実装する。
3	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。
4	jp.terasoluna.fw.batch.exception.IllegalClassTypeException	BatchUtil を使用して List から配列に変換する際に異常があった際にスローされる例外クラス。

◆ 拡張ポイント

なし

■ 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)

Copyright © 2011-2016 NTT DATA CORPORATION.

■ 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">
      … (省略) …
</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">
      … (省略) …
</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) …
```

Copyright © 2011-2016 NTT DATA CORPORATION.

✧ 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) {
        … (省略) …
    }
```

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。

Copyright © 2011-2016 NTT DATA CORPORATION.

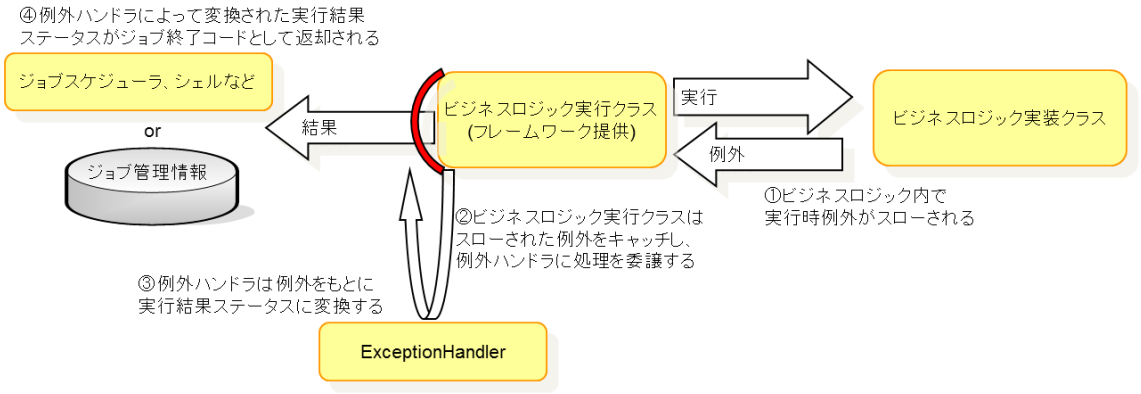
BL-04 例外ハンドリング機能

■ 概要

◆ 機能概要

- ビジネスロジック内でスローされた実行時例外を実行結果ステータスに変換する機能を提供する
- 例外ハンドラ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時例外がスローされる

② ビジネスロジック実行クラスはスローされた例外をキャッチし、例外ハンドラに処理を委譲する

③ 例外ハンドラは例外をもとに実行結果ステータスに変換する
使用される例外ハンドラは、Bean 定義ファイルにジョブ個別例外ハンドラを設定した場合は、ジョブ個別例外ハンドラとなる。実装されていない場合は、あらかじめブランクプロジェクトに定義されているジョブ共通のデフォルト例外ハンドラである DefaultExceptionHandler となる。

④ 例外ハンドラによって変換された実行結果ステータスがジョブ終了コードとして返却される

Copyright © 2011-2016 NTT DATA CORPORATION.

■ 使用方法

◆ コーディングポイント

【コーディングポイントの構成】

- Bean 定義ファイルの設定
  - デフォルト例外ハンドラの設定
  - 例外とジョブ終了コードの変換テーブル設定

- Bean 定義ファイルの設定

  - デフォルト例外ハンドラの設定

    業務処理で例外が発生した場合に 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 DefaultExceptionHandler(デフォルト例外ハンドラ)があらかじめ提供されている。

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

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

    DefaultExceptionHandler の処理内容では足りない場合は、ExceptionHandler インタフェースを実装する独自クラスで置き換えることができる。

    ✧ ジョブ共通例外ハンドラの実装例

```
public class CustomExceptionHandler implements ExceptionHandler {
    private static Logger log = LoggerFactory.getLogger(CustomExceptionHandler.class);

    @Inject
    MessageAccessor messageAccessor;

    @Override
    public int handleThrowableException(Throwable e) {
        … (省略) …
        // ジョブ終了コードに変換する
        return 100;
    }
}
```

    ✧ デフォルト例外ハンドラの置き換え例(beansDef/commonContext.xml)

```
<bean id="defaultExceptionHandler"
        class="jp.terasoluna.sample.xxx.CustomExceptionHandler" />
```

DefaultExceptionHandler を置き換える

Copyright © 2011-2016 NTT DATA CORPORATION.

➢ 例外とジョブ終了コードの変換テーブル設定

以下のように"exceptionToStatusMap"という識別子で例外と終了コードの変換テーブルを commonContext.xml に定義する。

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。

✧ 例外と終了コードの変換情報の設定例(beansDef/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。

■ 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.handler.BLogicExceptionHandlerResolver	例外ハンドラを解決するためのインタフェース。
4	jp.terasoluna.fw.batch.exception.handler.BLogicExceptionHandlerResolverImpl	例外ハンドラを解決するためにフレームワークが提供するデフォルトの実装クラス。
5	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。

Copyright © 2011-2016 NTT DATA CORPORATION.

◆ 拡張ポイント

- ジョブ個別例外ハンドラクラスの作成

ジョブごとにログ出力やジョブ終了コードへの変換ロジックを実装したい場合は、ジョブ個別例外ハンドラを作成する。

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ラクラスを「ジョブ業務コード」+「ExceptionHandler」という名前で作成して DI コンテナで管理しておくと、そのジョブに関してはデフォルト例外ハンドラの代わりに作成したジョブ個別例外ハンドラが呼び出される。

たとえば、ジョブ業務コード B000001 に対応するジョブ個別例外ハンドラクラスは「B000001ExceptionHandler」という名前になる。

✧ ジョブ個別例外ハンドラハンドラクラスの作成例

```
@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;
    }
}
```

ビジネスロジックと同じパッケージに配置して@Component アノテーションを付与すると Bean 定義を省略できる。

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

■ 関連機能

なし

■ 使用例

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

■ 備考

なし

Copyright © 2011-2016 NTT DATA CORPORATION.