TERASOLUNA

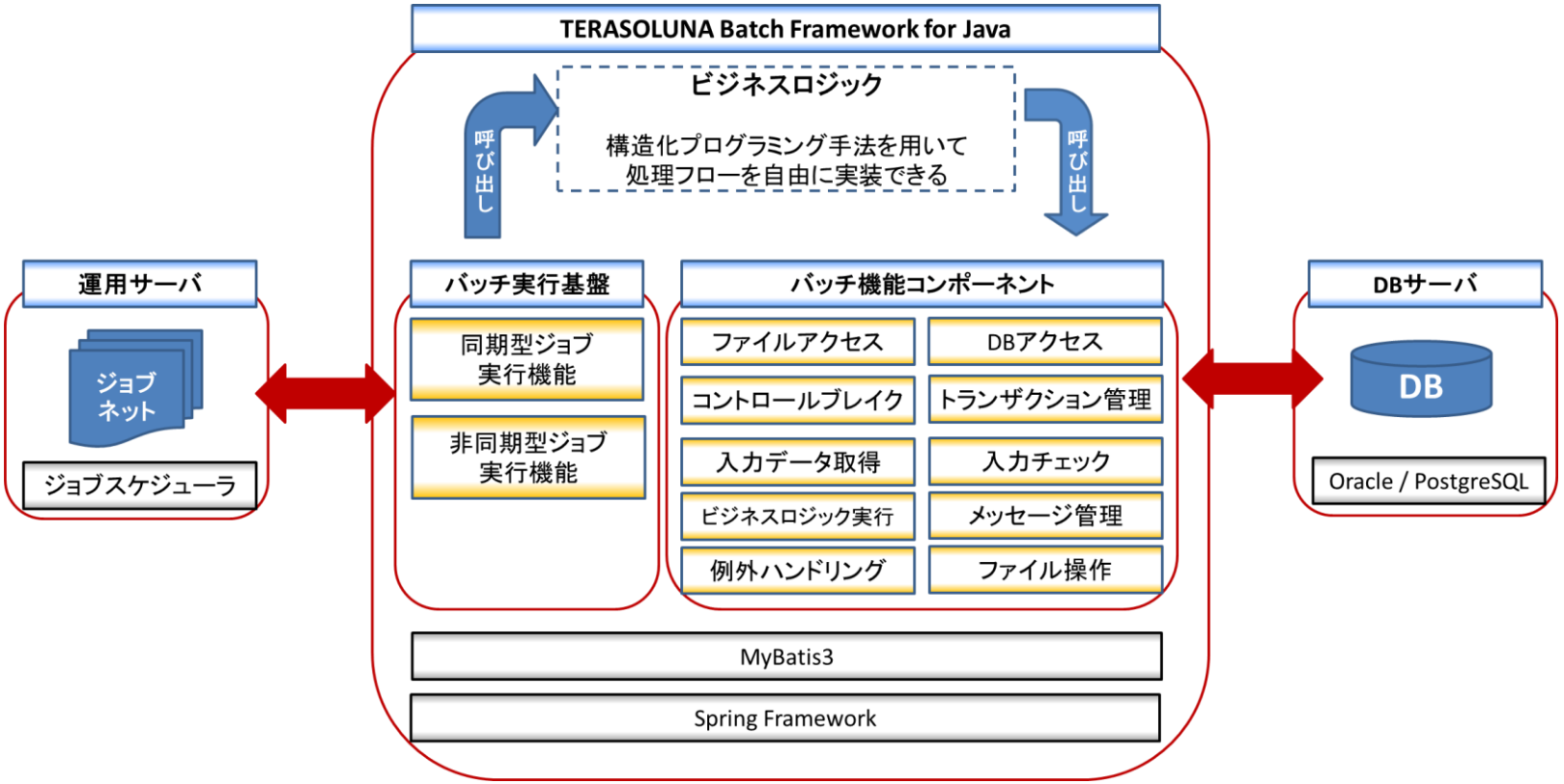
TERASOLUNA Batch Framework for Java Version 3.6.4 説明資料

株式会社NTTデータ
技術開発本部
ソフトウェア工学推進センタ

NTT DATA

TERAバッチ3.6.4アーキテクチャのコンセプト

- ①**オンラインの開発者は、すぐにでもバッチ開発を習得可能**です。
- ②バッチ開発に必要な機能を、**コンポーネント化して提供**しています。
- ③**構造化プログラミングでビジネスロジックを実装可能**であるため、以下の特徴があります。
- Pro*C、COBOLからのマイグレーションが容易です
- 処理設計書との親和性が高いです



コンポーネント仕様

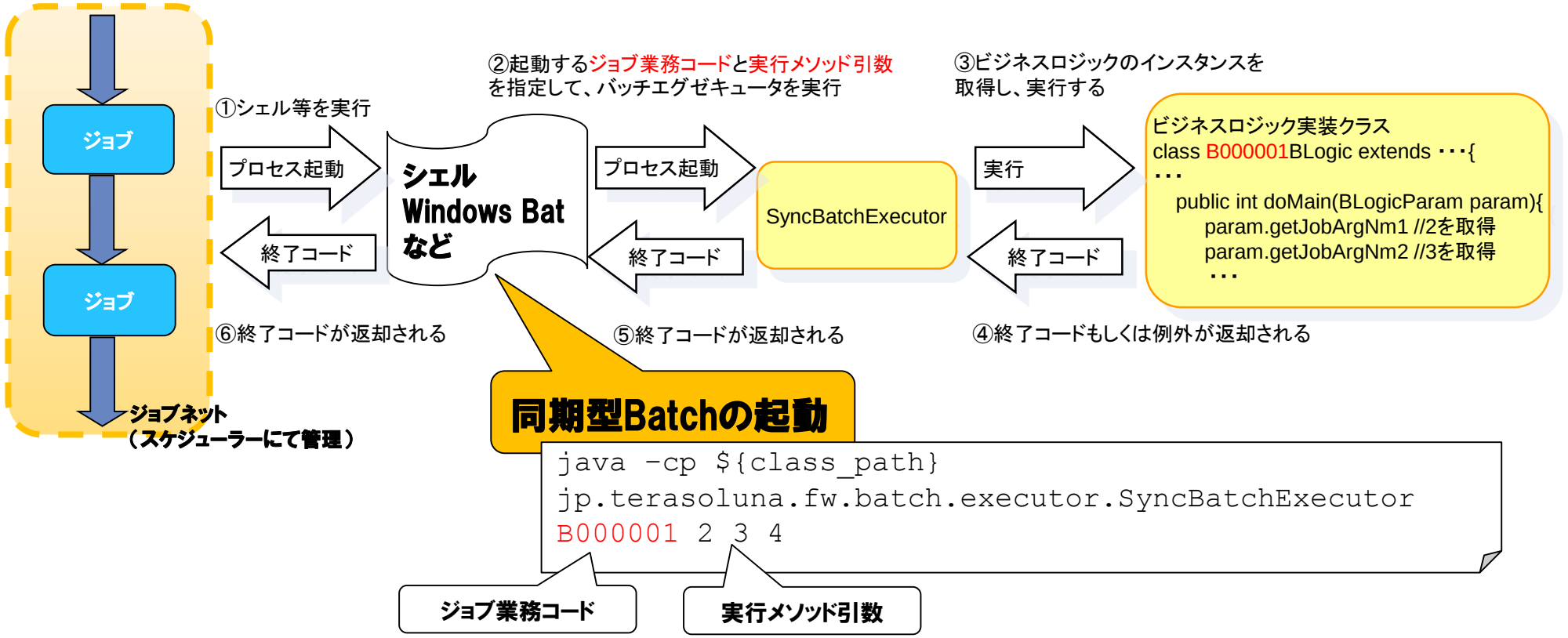
NTT DATA

- 各コンポーネントの概要を以下に示します。

機能	解説
同期型ジョブ実行	新規にプロセスを起動して、ジョブを実行する
非同期型ジョブ実行	ジョブ管理テーブルに登録されたジョブ情報をもとに、スレッドにてジョブを実行する
トランザクション管理	コミット、ロールバックなどのユーティリティメソッドを提供する
DBアクセス	MyBatis3を利用し、データベースアクセスを行う TERASOLUNA Server Framework for Java ver.5.xと同じものを利用する
ファイルアクセス	CSVや固定長ファイルを、オブジェクトにマッピングする機能
入力データ取得	データ収集を行うモジュールで、以下の特徴を持つ。 ・大量データ取得時にメモリを大量消費しない（フェッチサイズ分のみ） ・MyBatis3のResultHandlerを利用する場合と異なり、構造化プログラミング（while文）にて実装できる
メッセージ管理	プロパティファイルやDBに定義したメッセージを取得する機能 プレースホルダを利用し、文字列を自動的に置換することができる
ファイル操作	ファイルの削除、コピー、作成、マージ等の操作を行う機能
ビジネスロジック実行	BLogicクラスを実装するだけでビジネスロジックを実行できる機能
例外ハンドリング	ビジネスロジックで発生した例外をハンドリングする機能
入力チェック	アノテーションを利用した入力チェック機能
コントロールブレイク	現在読んだデータと、次に読むデータで、キーが切り替わるのを判定するユーティリティ

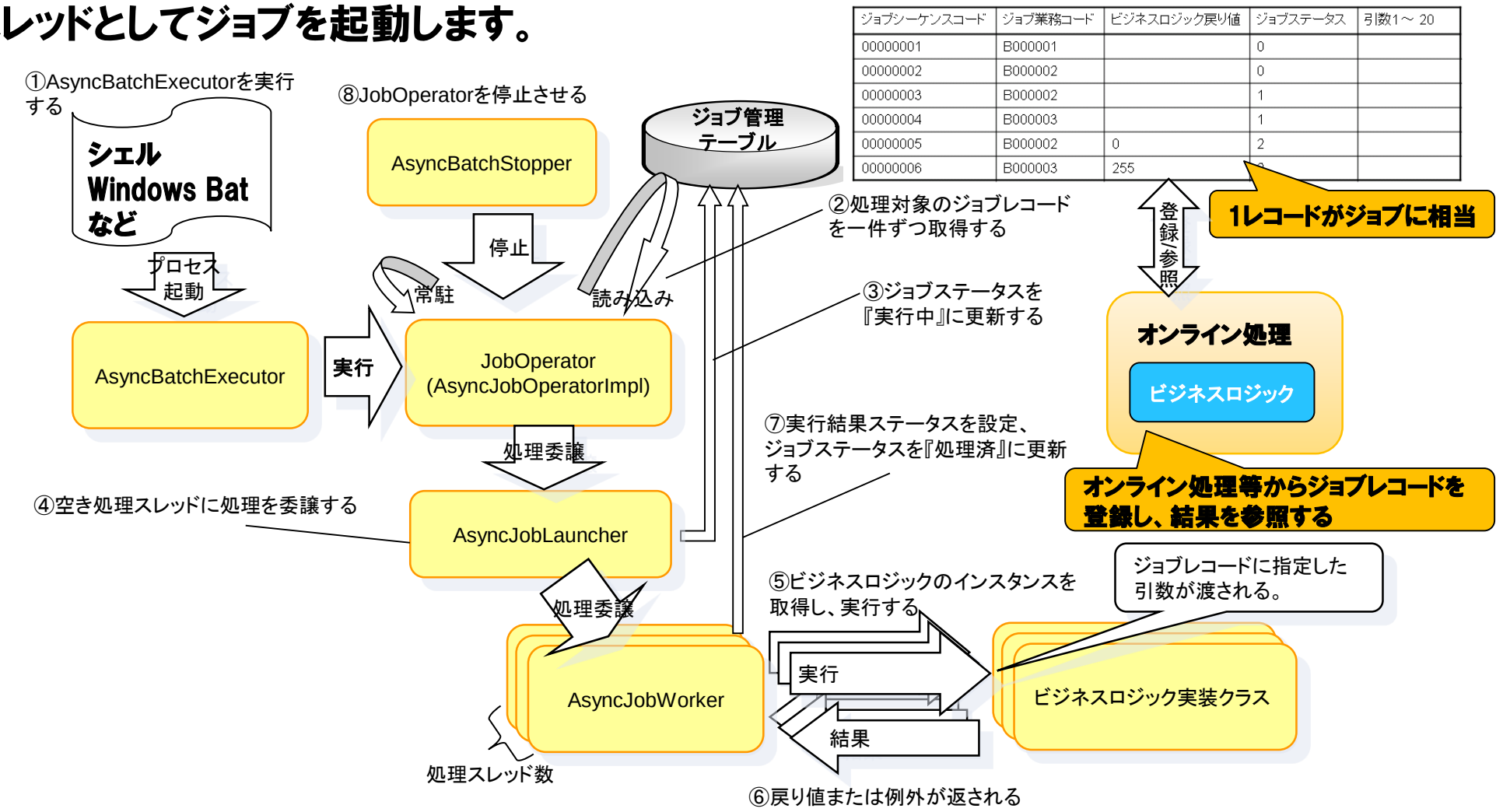
同期型ジョブ実行機能

同期型実行機能では、シェルからプロセスとして、バッチジョブを起動します。
シェル引数が、ビジネスロジック実装クラス(ジョブクラス)の実行メソッド引数に渡され、
メソッドの戻り値が、そのまま終了コードとして、シェルに戻されます。



非同期型ジョブ実行機能

非同期型実行機能では、「ジョブ管理テーブル」に登録された情報を元にして、スレッドとしてジョブを起動します。

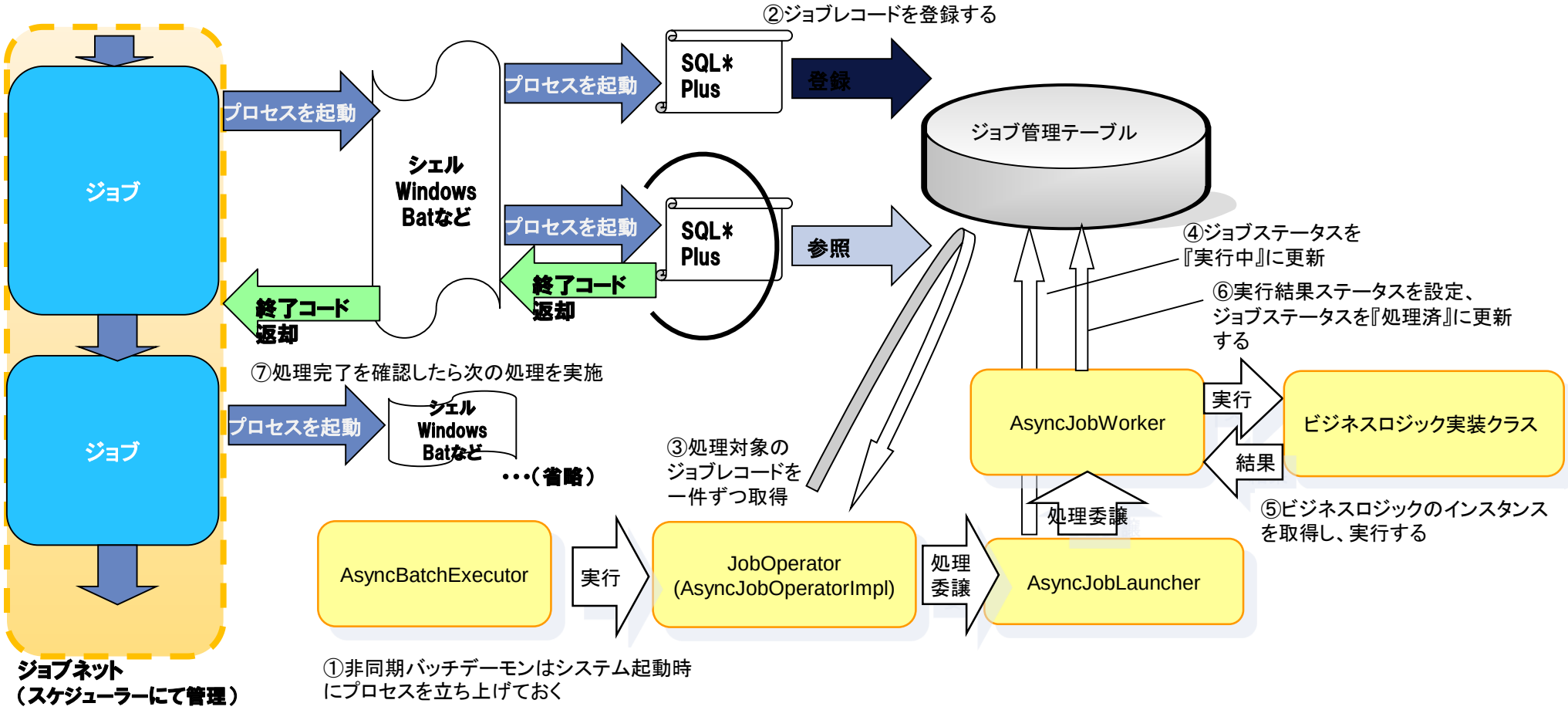


ジョブシーケンスコード	ジョブ業務コード	ビジネスロジック戻り値	ジョブステータス	引数1～20
00000001	B000001		0	
00000002	B000002		0	
00000003	B000002		1	
00000004	B000003		1	
00000005	B000002	0	2	
00000006	B000003	255		

【参考】非同期型ジョブ実行の応用例

処理時間が短いジョブ（1ジョブ数～数十秒）を連続して実行する場合に、非同期型ジョブ実行を利用することを推奨します。

（1ジョブごとにJavaプロセスの起動/終了をするとリソースを圧迫するため、上記の方式が望ましいです。）



実装イメージ①

構造化プログラミングの手法にて、ジョブ(＝BLogic)を作成します。
トランザクションはフレームワークが制御し、
「正常終了したらコミット」「例外が発生したらロールバック」
という動作をします。

ビジネスロジックコーディング例

トランザクション管理をフレームワークに任せる場合は、AbstractTransactionBLogicを継承する

```
...
@Component
public class B000002BLogic extends AbstractTransactionBLogic {
    private static Logger log = LoggerFactory.getLogger(B000002BLogic.class);

    @Inject
    B000002Dao b000002Dao = null;

    public int doMain(BLogicParam param) {
        InsertUser insertUser = new InsertUser();
        insertUser.setPassword("password");
        insertUser.setUserName(param.getJobArgNm1());

        b000002Dao.insertUser(insertUser);

        return 0;
    }
}
...
```

実装イメージ②

NTT DATA

- トランザクション管理をプログラマティックに実施することも可能です。

ビジネスロジックコーディング例

```
@Component
public class B000002BLogic  implements BLogic {
     ...
    @Inject
    PlatformTransactionManager transactionManager ;

    public int execute(BLogicParam param) {
        TransactionStatus stat = null;
        Collector<SampleData> collector = (略)
        try {
            SampleData inputData = null;
            stat = BatchUtil.startTransaction(transactionManager);
            int cnt = 0;
            while (collector.hasNext()) {
                cnt++;
                // DBへの更新処理
                ・・・省略
                if(cnt % 1000 == 0){
                    BatchUtil.commitTransaction(transactionManager, stat);
                    stat = BatchUtil.startTransaction(transactionManager);
                }
            }
            // 残りのデータのコミット
            BatchUtil.commitTransaction(transactionManager, stat);
        } catch (Exception e) {
            BatchUtil.rollbackTransaction(transactionManager, stat);
        ...
        } finally {
        ...
```

TransactionManagerのフィールドを定義する

トランザクション開始

1000件ごとにコミット

例外発生時はロールバック

実装イメージ（Bean定義ファイル）

- Bean定義ファイルは、「コンポーネントスキャン」を利用することで、
- 最小限の記載で済みます。

Bean定義ファイル名は
「**ジョブ業務コード**」+「.xml」
と設定する

ジョブBean定義の設定例

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

  <!-- 共通コンテキスト（フレームワークの共通機能を使う場合、必ずインポートすること。） -->
  <import  resource="commonContext.xml" />

  <!-- データソース設定 -->
  <import  resource="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" />
```

データベースアクセス機能を使用する場合に追加する

コンポーネントスキャナのベースパッケージに
業務のパッケージを指定する

DBアクセス①

- MyBatis3を利用して、データベースアクセスを行います。
- TERASOLUNA Server Framework for Java Version5.xと同じものを利用します

Bean定義ファイル名は
「**ジョブ業務コード**」+「.xml」
と設定する

DAOをBean定義する。

ジョブBean定義の設定例

```
・・・
<bean id="sampleDao" class="org.mybatis.spring.mapper.MapperFactoryBean">
    <property name="mapperInterface" value="jp.terasoluna.batch.sample.b000001.SampleDao" />
    <property name="sqlSessionTemplate" ref="sqlSessionTemplate" />
</bean>
```

マッピングファイル名は
「DAOインタフェースのFQCN」+「.xml」
と設定する

DAOインタフェースのFQCNを
namespaceに指定する。

取得するためのメソッド名をid属性に指定し、メソッド呼び出し時に発行するSQLを記載する。

マッピングファイルの設定例

```
<mapper namespace="jp.terasoluna.batch.sample.b000001.SampleDao">

  <select id="selectUser" parameterType="jp.terasoluna.batch.sample.b000001.SelectUserInputDto"
   resultType="jp.terasoluna.batch.sample.b000001.SelectUserOutputDto">
      SELECT
          USER_NAME as name,
          USER_ID as id
      FROM
          USER_TABLE
      WHERE
          USER_ID = #{id}
  </select>
```

DBアクセス②

ビジネスロジックの実装例

```
@Component
public class SampleLogic extends AbstractTransactionBLogic {

    @Inject
    SampleDao sampleDao = null; // インタフェースの型でDAOを宣言

    public int doMain(BLogicParam param) {
        SelectUserInputDto bean = new SelectUserInputDto(); //プレースホルダ置換用のPOJO
        ・・・省略
        SelectUserOutputDto result = sampleDao.selectUser(bean);
    }
}
```

データベースアクセス処理が行われ、SQLの実行結果をJavaBeanにマッピングされた形で取得できる。

DAOの作成例

```
public interface SampleDao {
    // SQLを呼び出すメソッドを定義
    SelectUserOutputDto selectUser(SelectUserInputDto selectUserInputDto);

}
```

メソッド名とマッピングファイルのid属性に指定した文字列が同じSQLを呼び出す。

ファイルアクセス

- 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します。

	インタフェース名	説明
1	FileQueryDAO / FileUpdateDAO	ファイル入力用DAOインタフェース / ファイル出力用DAOインタフェース
2	FileLineIterator / FileLineWriter	ファイル入力用イテレータ / ファイル出力用ライタ

	クラス名	説明
1	CSVFileQueryDAO / CSVFileUpdateDAO	CSV形式のファイル入力（出力）を行う場合に利用する
2	FilexedFileQueryDAO / FixedFileUpdateDAO	固定長形式のファイル入力（出力）を行う場合に利用する
3	VariableFileQueryDAO / VariableFileUpdateDAO	可変長形式（タブ区切り等）のファイル入力（出力）を行う場合に利用する

	クラス名	説明
1	CSVFileLineIterator / CSVFileLineWriter	CSV形式のファイル入力（出力）を行う場合に利用する
2	FilexedFileLineIterator / FixedFileLineIterator	固定長形式のファイル入力（出力）を行う場合に利用する
3	VariableFileLineIterator / VariableFileLineWriter	可変長形式（タブ区切り等）のファイル入力（出力）を行う場合に利用する

ファイルアクセス（続き）

「ビジネスロジック」の実装例

```
@Inject
@Named(value = "csvFileQueryDAO")
FileQueryDAO csvFileQueryDAO = null;

public int execute(BLogicParam param) {
  ・・・
  // ファイル入出力用イテレータの取得
  FileLineIterator<SampleFileLineObject> fileLineIterator
    = csvFileQueryDAO.execute(basePath +
      "/some_file_path/uriage.csv", FileColumnSample.class);
  try {
    // ヘッダ部の読み込み
    List<String> headerData =
          fileLineIterator.getHeader();
    ・・・ // 読み込んだヘッダ部に対する処理
    while(fileLineIterator.hasNext()){
      // データ部の読み込み
      SampleFileLineObject sampleFileLine =
                                fileLineIterator.next();
      ・・・ // 読み込んだ行に対する処理
    }
    // トレイラ部の読み込み
    List<String> trailerData =
                      fileLineIterator.getTrailer();
    ・・・ // 読み込んだトレイラ部に対する処理
  } finally {
    // ファイルのクローズ
    fileLineIterator.closeFile();
  }
  ...
}
```

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

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

ファイル形式に関わらず、next()メソッドを使用する

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

「ファイル行オブジェクト」の実装例

```
@FileFormat
public class SampleFileLineObject {

    @InputFileColumn(
         columnIndex = 0,
         columnFormat=”yyyy/MM/dd”)
    private Date hiduke = null;

    @InputFileColumn(
            columnIndex = 1,
            stringConverter =
                StringConverterToUpperCase.class)
    private String shopId = null;

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

「入力ファイル」の例

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

入力データ取得（コレクタ）

- 入力データの取得は、入力データ取得機能（コレクタ）を利用して実施します。
- コレクタは以下の特徴を持ちます。
  - 大量データ取得時にメモリを大量消費しません（フェッチサイズ分のみ）。
  - MyBatis3のResultHandlerのみを利用して実装した場合と異なり、構造化プログラミング（while文）にて実装できます。

```
@Component
public class SampleBLogic extends AbstractTransactionBLogic{
    ・・・省略
    public int doMain(BLogicParam param){
        Collector<UserBean> collector = new DaoCollector<UserBean>(
                    this.sampleDao, "collectUserList", null);
        try{
            UserBean bean = null;
            while(collector.hasNext()){
                //入力データを取得
                bean = collector.next();
                //入力データに対する処理を記述する
                 ・・・省略
            }
        }finally{
           CollectorUtility.closeQuietly(collector);
        }
        ・・・省略
    }
    ・・・省略
}
```

入力チェック

- Bean Validationのバリデータを利用します。
- 入力チェックはコレクタ内で行われます。

```
public class UserBean {
    // nameプロパティが、nullを許容せず、
    // 1文字以上20文字以下の場合の設定例
    @NotNull
    @Size(min=1, max=20)
    private String name;

    // ageプロパティが1以上、200以下の場合の設定例
    @Min(1)
    @Max(200)
    private int age;
    // setter, getterは省略するが必要
}
```

ルール名	概要
NotNull	nullでないこと
Null	nullであること
Patter	正規表現にマッチすること
Min	指定された値以上であること
Max	指定された値以下であること
DecimalMin	Decimal型の値が指定された値以上であること
DecimalMax	Decimal型の値が指定された値以下であること
Size	文字列のlengthがminとmaxの間であること
Digits	指定された範囲内の数値であること
AssertTrue	trueであること
AssertFalse	falseであること
Future	未来日付であること
Past	過去日付であること
CreditCardNumber	Luhnアルゴリズムに準拠したクレジットカード番号であること
Email	RFC2822に準拠していること
URL	RFC2396に準拠していること
NotBlank	null, 空文字, 空白のみでないこと
NotEmpty	null, または空でないこと

```
@Inject
@Named("beanValidator")
Validator validator;

public int doMain(BLogicParam param) {
    //コレクタ生成
    Collector<UserBean> collector =
        new DaoValidateCollector<UserBean>(
            this.sampleDao, "colletData01",
            null, validator);

    try {
        UserBean bean = null;
        while (collector.hasNext()) {
        // データの取得
        bean = collector.next();
        ・・・省略
    }
}
```

DaoValidateCollector(ファイルに対して入力チェックを行う場合はFileValidateCollector)を生成する。第四引数に入力チェックを行うValidatorクラスを渡す。

nextメソッド実行時に次の対象データの入力チェックが行われる。
入力チェックエラー発生時、拡張入力チェックエラーハンドラを使用しない場合は、「ValidationErrorException」がスローされる。

コントロールブレイク

- 「キーの切り替わり」を検知するユーティリティ（ControlBreakChecker）を提供します。

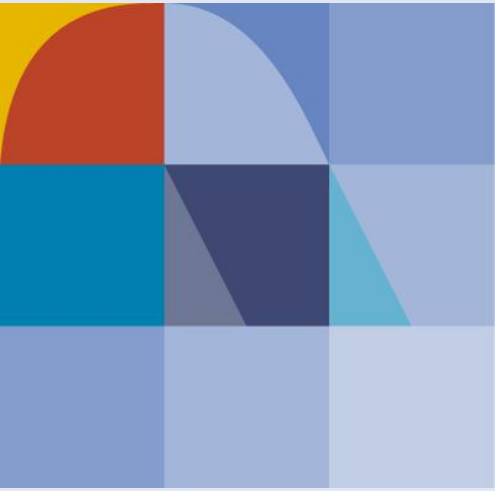
ブレイク処理（キーが支社のとき）

```
支社,担当者,請求書番号
千葉,田中,100001
千葉,田中,100002
千葉,田中,100003
千葉,佐藤,100004
千葉,佐藤,100005 ←ここでブレイク処理
埼玉,鈴木,100006
```

ControlBreakCheckerの主なメソッド

ControlBreakCheckerのメソッド	戻り値	概要
isBreak(Collector collector, String key)	boolean	keyのカラムに対し、現在の値と一つ後の値を比較し、値が切り替わっている(後ブレイク)場合はtrueを返す
getBreakKey(Collector collector, String key)	Map<String, Object>	keyのカラムに対し後ブレイクが発生した際に、キーの切り替わりが発生したカラム名と値のマップを返す
isPreBreak(Collector collector, String key)	Boolean	keyのカラムに対し、現在の値と一つ前の値を比較し、値が切り替わっている(前ブレイク)場合はtrueを返す
getPreBreakKey(Collector collector, String key)	Map<String, Object>	keyのカラムに対し前ブレイクが発生した際に、キーの切り替わりが発生したカラム名と値のマップを返す

```
Collector<UserBean> collector = （略）
try {
    UserBean bean = null;
    while  ( collector.hasNext()) {
        //入力データを取得
        bean = collector.next();
        //入力データに対する処理を記述する
        ・・・
        //キーの切り替わりの検知
        if (ControlBreakChecker.isBreak(collector , "sisya")){
            //ブレイク発生時の処理を記述する
        }
    }
} finally {
    CollectorUtility.closeQuietly(collector);;
}
```



NTT DATA

Global IT Innovator