TERASOLUNA

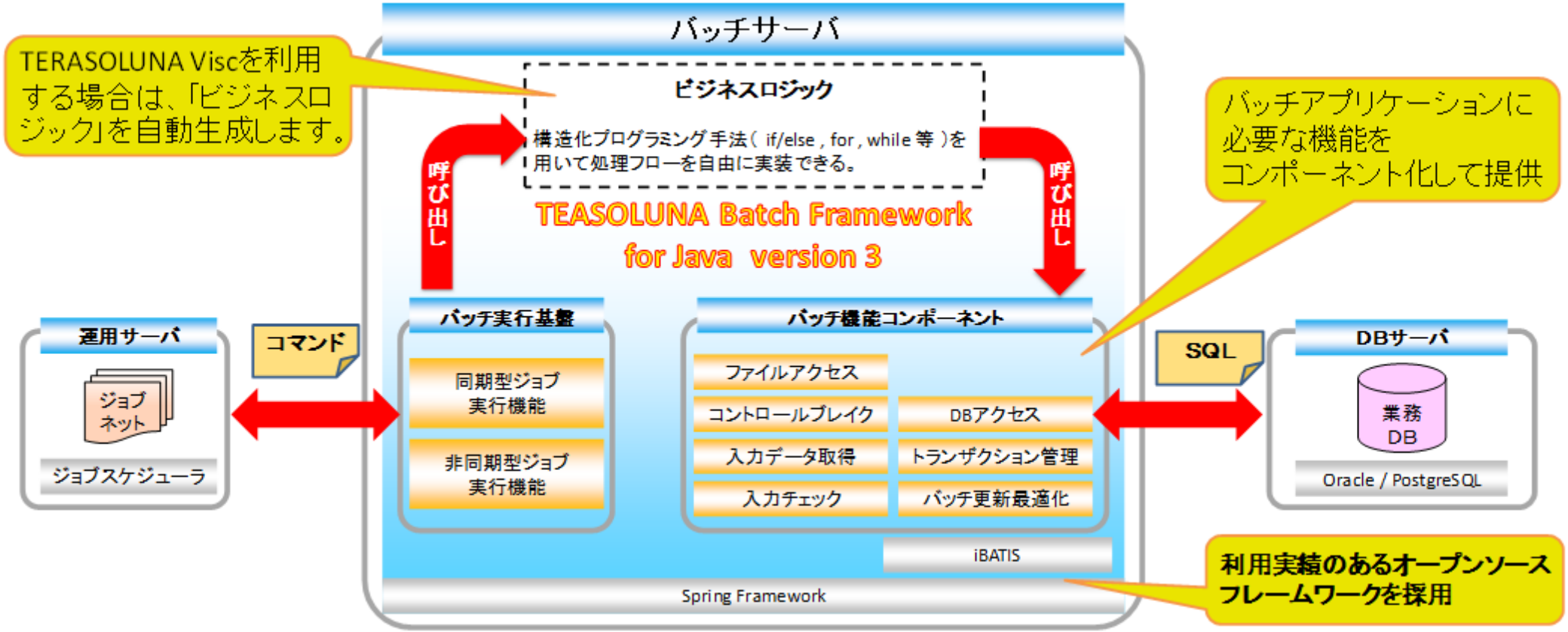
TERASOLUNA Batch Framework for Java Version 3.x 説明資料

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.x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
- Viscに採用されています



コンポーネント仕様

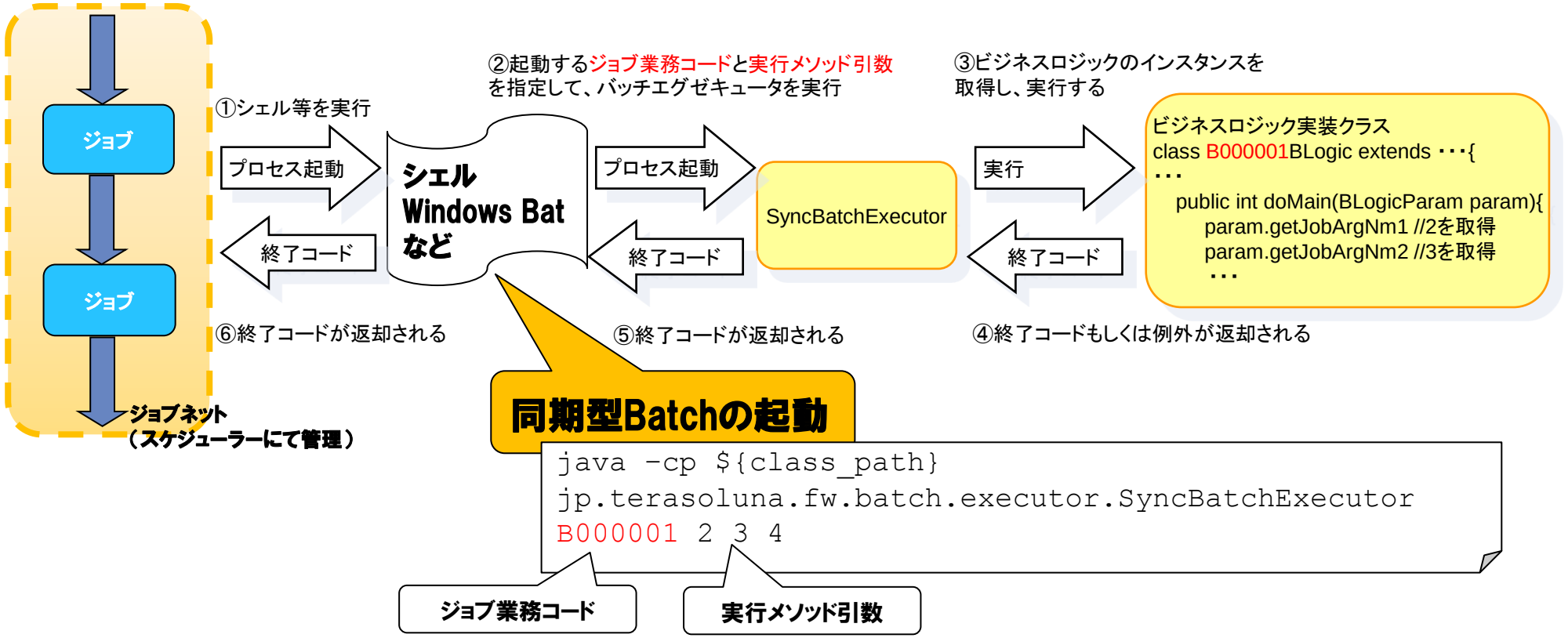
NTT DATA

- 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	iBATISを利用した、ORマッピング機能（TERASOLUNAの、QueryDAO、UpdateDAO、QueryRowHandleDAOなどをそのまま利用）
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（現行バッチFW（Version2）の、FileDAOをそのまま利用）
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） ・QueryRowHandleDAOと異なり、構造化プログラミング（while文）にて実装できる
入力チェック	設定ファイルベースのバリデータ（TERASOLUNA Server Framework for Java (Rich版) で利用しているバリデータ）
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
バッチ更新最適化	一括して同種のSQLを実行することでスループット向上を実現するしくみ

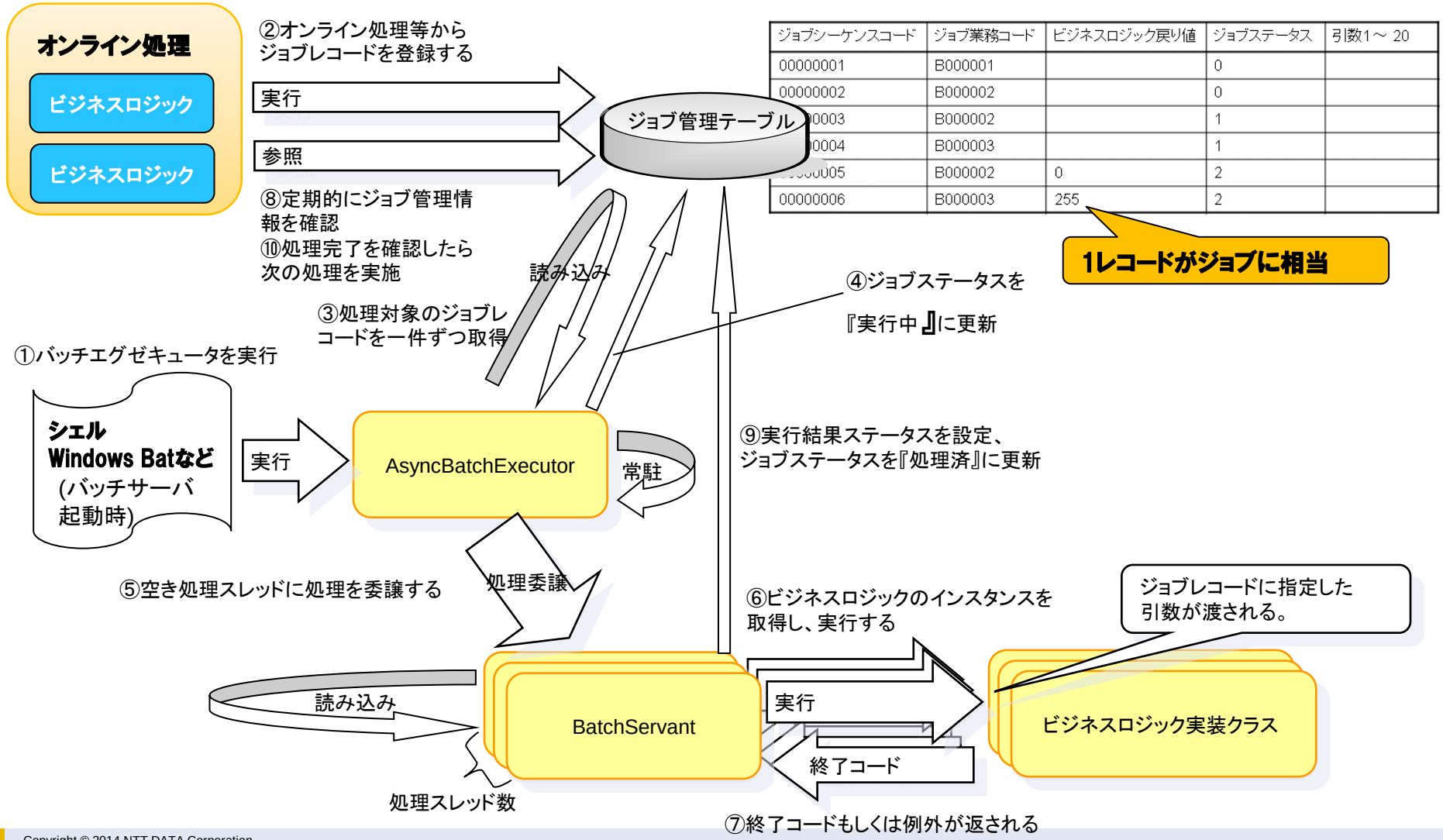
同期型ジョブ実行機能

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。

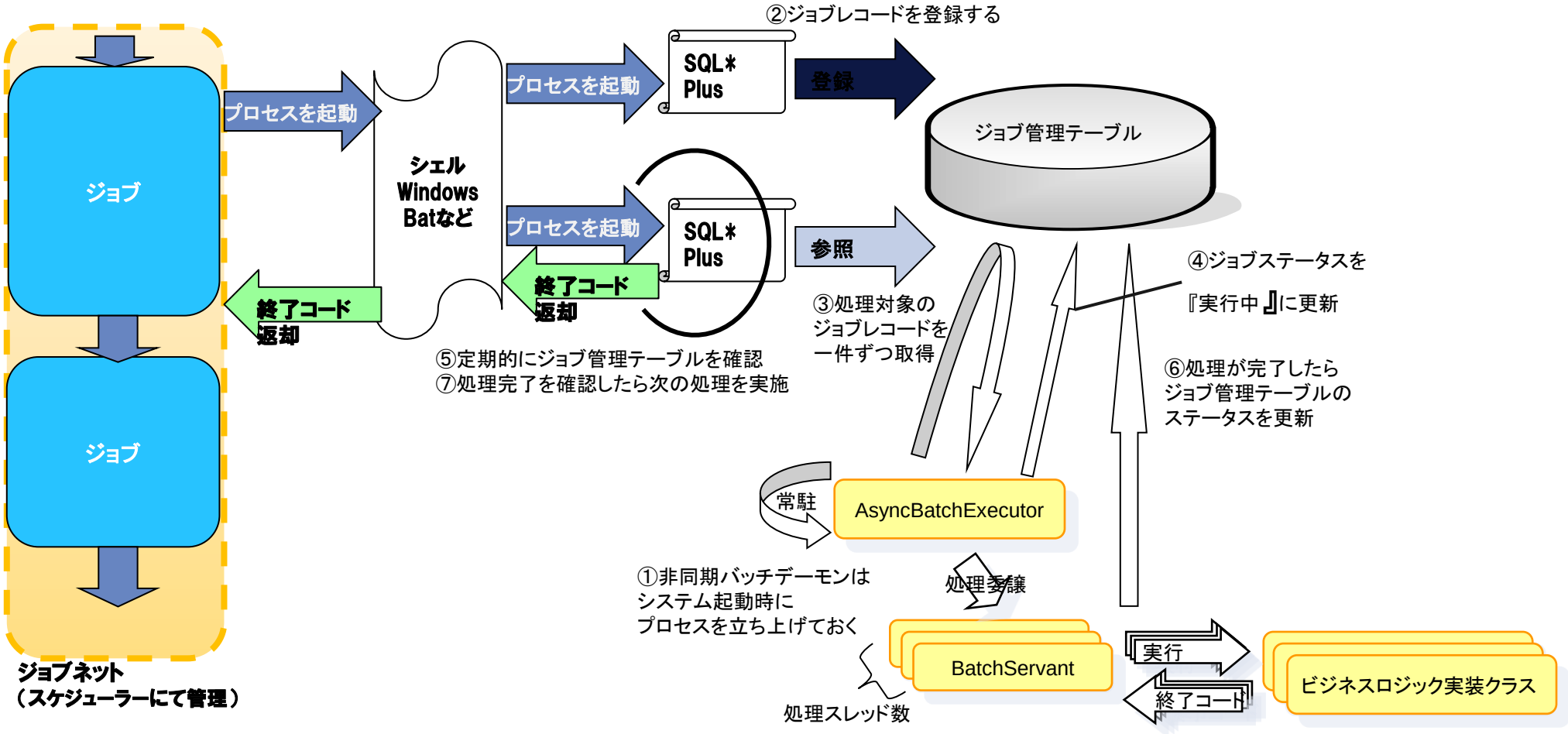


ジョブシーケンスコード	ジョブ業務コード	ビジネスロジック戻り値	ジョブステータス	引数1～ 20
00000001	B000001		0	
00000002	B000002		0	
00000003	B000002		1	
00000004	B000003		1	
00000005	B000002	0	2	
00000006	B000003	255	2	

【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

```
・・・
@Component
public class B000002BLogic extends AbstractTransactionBLogic {
    private static Log log = LogFactory.getLog(B000002BLogic.class);

    @Autowired
    protected UpdateDAO updateDAO = null;

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

        updateDAO.execute("b000002.insertUser", insertUser);

        return 0;
    }
}
・・・
```

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

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 {
    ・・・
  @Autowired
  protected  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。

ジョブBean定義の設定例

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

```
・・・

   <!-- 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、かならずインポートすること。) -->
   <import  resource="commonContext.xml" />

   <!-- データソース設定1  -->
   <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" />

・・・
```

QueryDAOやUpdateDAO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

- iBATISを利用して、データベースアクセスを行うコンポーネントです。
- TERASOLUNA Server/Batch Framework for Java Version2.xと同じものを利用します。
  - QueryDAO・・・参照系のデータベースアクセスを行うDAO
  - UpdateDAO・・・更新系のデータベースアクセスを行うDAO
  - StoredProcedureDAO・・・ストアドプロシージャを実行するDAO
  - QueryRowHandleDAO・・・参照系のデータベースアクセスの結果を1件ずつ処理するためのDAO

    ※大量データの読み込みを行う場合は、「コレクタ」を同時に利用します。

```
@Component
public class QueryBLogic extends AbstractTransactionBLogic {
    @Autowired
    protected QueryDAO queryDAO = null; // インタフェースの型でDAOを宣言

    public int doMain(BLogicParam param) {
        SelectUserArgBean bean = new SelectUserArgBean(); //プレースホルダ置換用のPOJO
        ・・・省略
        SelectUserResult result =
            queryDAO.queryForObject("select.user", bean, SelectUserResult.class);
        //第一引数は実行する設定のID(DAOの実装に依存する)
        //第二引数はSQLなどにプレースホルダがある場合の置換文字列を格納したPOJO。不要であればnull。
        //第三引数は戻り値のクラス
        ・・・省略
    }
}
```

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。
- TERASOLUNA Batch Framework for Java Version2.xと同じものを利用します。

	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

```
    @Aurtowired
    @Qualifier(value = "csvFileQueryDAO")
    private FileQueryDAO csvFileQueryDAO = null;
...
    // ファイル入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

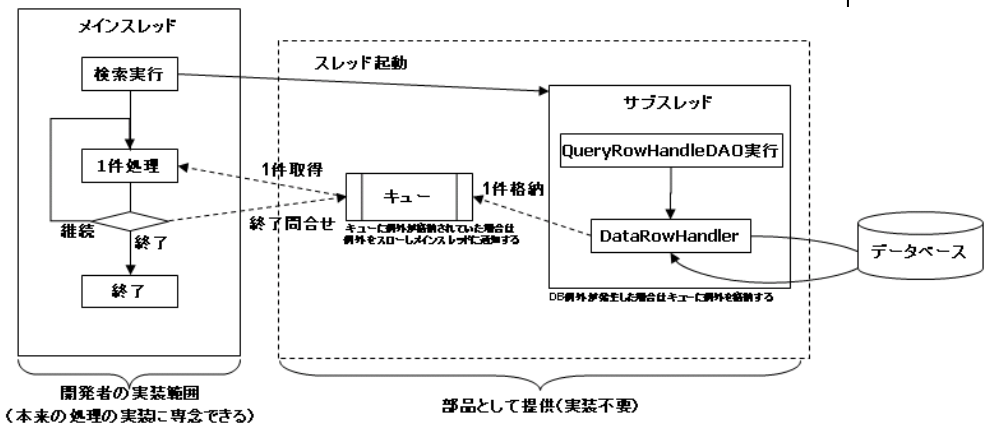
```
“2006/07/01”,”shop01”,”1,000,000”
```

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）。
  - QueryRowHandleDAO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 DBCollector<UserBean>(
                    this.queryRowHandleDAO, "selectUserList", 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
```

コレクタの内部構造



入力チェック

- TERASOLUNA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。

```
<form-validation>
  <formset>
    <form name="jp.terasoluna.batch.sample.b000010.dto.CsvRecord">
      <field property="id" depends="required">
        <arg key="id要素" position="0"/>
      </field>
      <field property="num" depends="required">
        <arg key="num要素" position="0"/>
      </field>
      <field property="name" depends="required">
        <arg key="name要素" position="0"/>
      </field>
      <field property="old" depends="required">
        <arg key="old要素" position="0"/>
      </field>
    </form>
  </formset>
</form-validation>
```

```
@Autowired
@Qualifier(value = "beanValidator")
private Validator validator;

public int doMain(BLogicParam param) {
    // コレクタ生成
    Collector<UserBean> collector = new DBValidateCollector<UserBean>(
        this.queryRowHandleDAO, "Sample.selectData01", null, validator);

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

        ...
    }
...
}
```

DBValidateCollector(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BeanValidatorクラスを渡す。

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

ルール名	概要
required	必須入力チェック
mask	正規表現一致チェック
byte	byte型チェック
short	short型チェック
integer	int型チェック
long	long型チェック
float	float型チェック
double	double型チェック
date	日付形式チェック
intRange	int型範囲チェック
doubleRange	double型範囲チェック
floatRange	float型範囲
maxLength	最大文字数制限
minLength	最小文字数制限
alphaNumericString	半角英数字文字列チェック
hankakuKanaString	半角カナ文字列チェック
hankakuString	半角文字列チェック
zenkakuString	全角文字列チェック
zenkakuKanaString	全角カナ文字列チェック
capAlphaNumericString	大文字半角英数字文字列チェック
number	数値チェック
numericString	数字文字列チェック
prohibited	禁止文字列チェック
stringLength	文字列長チェック
byteRange	byte列長範囲チェック
dateRange	date型範囲チェック
arrayRange	配列要素数チェック
url	URL形式チェック

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
```

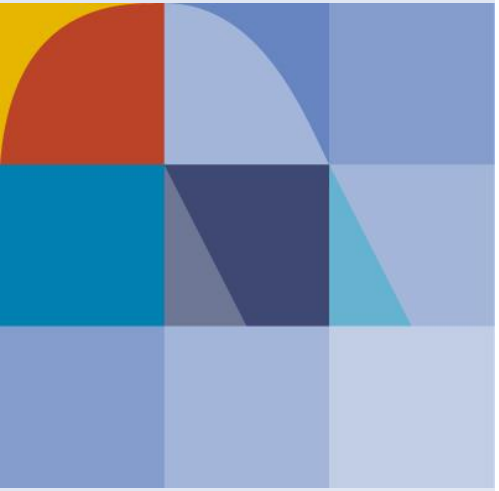
```
public boolean isBreak (Collector collector , String key) ;
public boolean isBreak (Collector collector , String [] keys) ;
public java.util.Map checkBreak (Collector collector , String [] keys) ;
public boolean isPreBreak (Collector collector , String key) ;
public boolean isPreBreak (Collector collector , String [] keys) ;
public java.util.Map checkPreBreak (Collector collector , String [] keys) ;
```

```
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

変える力を、ともに生み出す。