AL-041　入力データ取得機能(コレクタ)

■ 概要

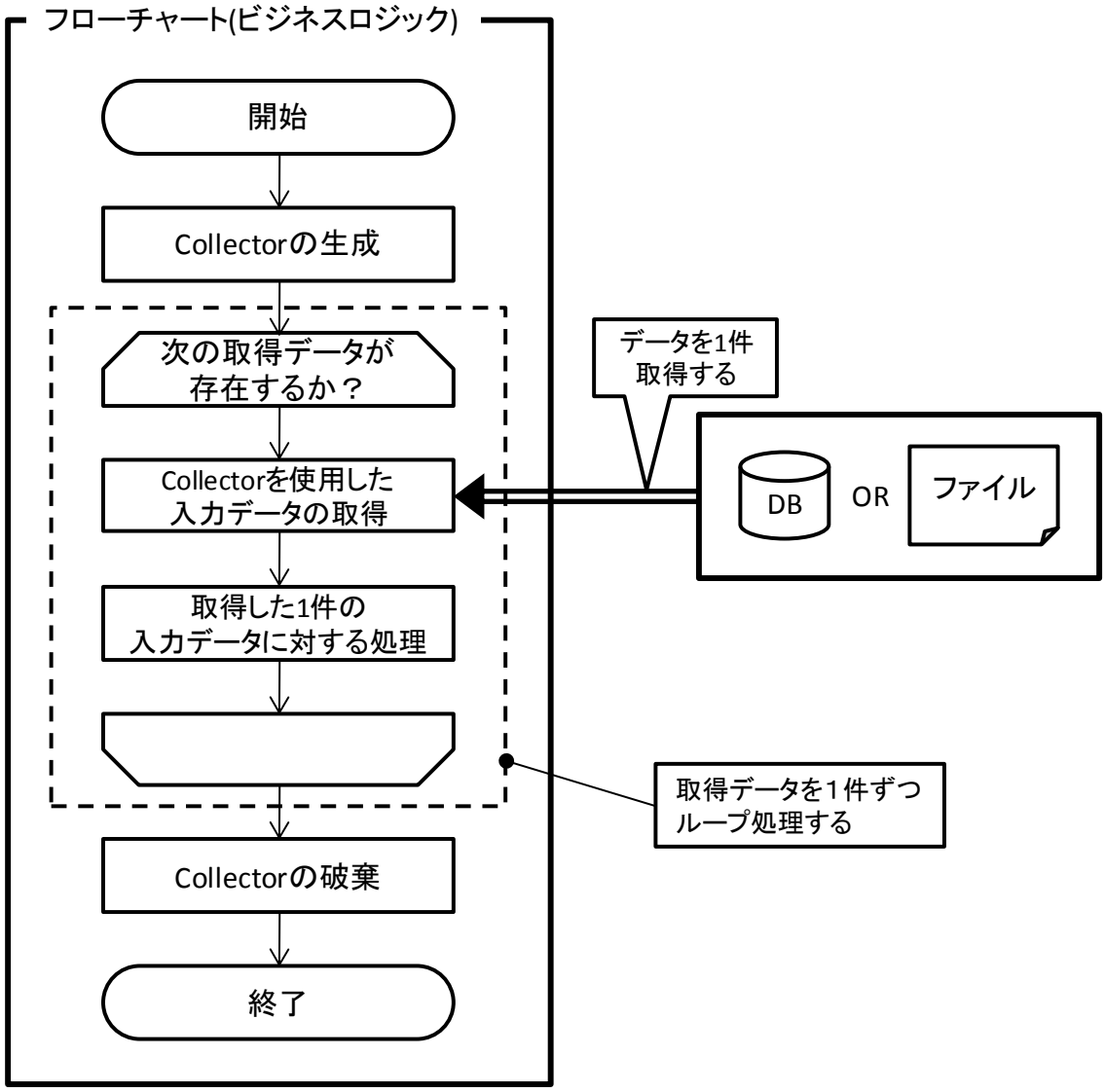
◆ 機能概要

- DB やファイルから入力データを取得する際に使用する機能である。
- DB からデータを取得する際、ResultHandler をメソッドの引数に持つ DAO を使用し、データを 1 件ずつ取得する。
- ResultHandler をメソッドの引数に持つ DAO を使用しない場合と比べ、ヒープメモリの消費量を抑制することができる。
- ファイルからのデータ取得時には FileQueryDAO を使用し、ファイルを 1 行ずつ取得する。
- 入力データ取得機能を使用することにより、ビジネスロジックを構造化プログラミングで作成できる。

◆ 注意点

- コレクタによるデータの取得は、ビジネスロジックとは別のトランザクションで実行される。よって、DB の更新処理を実施し、コミットする前にコレクタを用いて、更新処理したデータを取得しようとした場合に、更新処理後のデータが取得できないので注意する必要がある。詳細は備考を参照のこと。

◆ 概念図



◆ 解説

- ビジネスロジックで、DB やファイルからデータを 1 件毎に取得する。
  - ビジネスロジックでは、初めに Collector インスタンスを生成し、以降はループ処理の中で取得したデータに対する処理を記述する。
  - データ取得部分を担当する Collector クラスは本機能が提供している。
  - DB から大量データを処理する場合であっても、ビジネスロジックではデータを 1 件ずつ扱うため、ヒープ領域の圧迫を抑制することができる。
    結果、取得するデータ量に因る性能への影響を極力抑えることができる。

◆ コーディングポイント

【コーディングポイントの構成】

- データベースからデータを取得する場合
  - DAO への ResultHandler を引数に持つメソッドの定義
  - ビジネスロジックの実装
  - 例外処理
- ファイルからデータを取得する場合
  - ビジネスロジックの実装
  - 例外処理
- Collector クラスのコンストラクタについて
  - コンストラクタで設定できる内容について
  - Collector のコンストラクタ一覧
  - コンストラクタ引数一覧

➢ データベースからデータを取得する場合

データベースからデータを取得する場合、ビジネスロジックを実装する以外に、ResultHandler を引数に持つメソッドを DAO に新しく定義する必要がある。ResultHandler は MyBatis3 がデータベースから取得したデータを 1 件ずつ処理する際に使用するインタフェースである。

➢ DAO への ResultHandler を引数に持つメソッドの定義

DAO の実装方法は、『BL-06 データベースアクセス機能』を参照すること。入力データ取得機能を使用する際は、以下のようにメソッドを定義しなければならない。

✧ DAO のメソッド定義例

```
public interface Sample01Dao {

    // ResultHandler を引数に持つメソッド定義
    void collectData01(Object inputDto, ResultHandler<Data01Dto> handler);

    // DB 更新する通常のメソッド定義
    int updateData01(Data01Dto data01Dto);

}
```

入力データ取得機能で使用するメソッド定義は、必ず以下のシグネチャで定義する。戻り値、引数の型、及び引数の順番を変更することはできない。
戻り値:(型)void 型
第 1 引数:(型)Object, (引数)SQL にバインドされるパラメータ
第 2 引数:(型)ResultHandler インタフェース, (引数) ResultHandler の実装クラス
※TERASOLUNA Batch3.6.x でサポートする MyBatis3.3.0 以降では、ResultHandler に型引数を指定する。

➢ ビジネスロジックの実装

以下に入力データ取得機能を利用して、データベースからデータを取得する例を掲載する。

✧ ビジネスロジックの実装例

```
@Component
public class Sample01BLogic extends AbstractTransactionBLogic {

    @Inject
    Sample01Dao sample01Dao;

    @Override
    public int doMain(BLogicParam param) {

        // Collector の生成
        Collector<Sample01Bean> collector = new DaoCollector<Sample01Bean>(
                this.sample01Dao, "collectData01", null);

        try {
            Sample01Bean inputData = null;
            while (collector.hasNext()) {
                // データの取得
                inputData = collector.next();

                // ファイルの出力など、取得データに対する処理を記述する(実装は省略)
            }
        } catch (Exception e) {
            // 例外処理
        } finally {
            // Collector のクローズ
            CollectorUtility.closeQuietly(collector);
        }
        return 0;
    }
}
```

コンストラクタを使ってコレクタを生成する。
第 1 引数: DAO インスタンス※1
第 2 引数:データを取得するメソッド名
第 3 引数:SQL にバインドされるパラメータ
※1：ResultHandler をメソッドの引数に持った DAO を使用する。

while 文を使用して次のデータが存在する限り取得データに対してループ処理を行う。

必ず処理の最後にコレクタをクローズすること

➢ 例外処理

データベースアクセス時に例外が発生した場合、Collector#next メソッドを実行した際に発生した例外(MyBatisSystemException や BadSqlGrammarException など、DataAccessException のサブクラス)がビジネスロジックにスローされる。SQL 実行時エラー発生後に、後続の処理を継続することはできない。

➢ ファイルからデータを取得する場合

ファイルアクセスを行うためにフレームワークが提供するファイル入力用 DAO の使用方法や、ファイル行オブジェクトの実装方法は、『BL-07 ファイルアクセス機能』を参照すること。ファイルからデータを取得する場合はビジネスロジックの実装のみを行えば良い。

➢ ビジネスロジックの実装

以下に入力データ取得機能を利用して、ファイルからデータを取得する例を掲載する。

✧ ビジネスロジックの実装例

```
@Component
public class Sample02BLogic extends AbstractTransactionBLogic {

    @Inject
    @Named("csvFileQueryDAO")
    FileQueryDAO csvFileQueryDao;

    @Override
    public int doMain(BLogicParam param) {

        // Collector の生成
        Collector<Sample02Bean> collector = new FileCollector <Sample02Bean>(
                this.csvFileQueryDao, "inputFile/sinput_Sample02.csv", Sample02Bean.class);

        try {
            Sample02Bean inputData = null;
            while (collector.hasNext()) {
                // データの取得
                inputData = collector.next();

                // DB の更新など、取得データに対する処理を記述する(実装は省略)
            }
        } catch (Exception e) {
            // 例外処理
        } finally {
            // Collector のクローズ
            CollectorUtility.closeQuietly(collector);
        }
        return 0;
    }
}
```

CSV ファイルからのデータ取得時は CSVFileQueryDAO を使用する

コンストラクタを使ってコレクタを生成する。
第 1 引数: FileQueryDAO
第 2 引数:読み込むファイル名
第 3 引数:ファイル行オブジェクト

while 文を使用して次のデータが存在する限り取得データに対してループ処理を行う。

データ取得時に発生した例外は、ここでキャッチされる。

必ず処理の最後にコレクタをクローズすること

➢ 例外処理

ファイルアクセス時に例外が発生した場合、Collector#next メソッドを実行した際に発生した例外(ファイルアクセス機能がスローした例外)がビジネスロジックにスローされる。以下に FileException(ファイルアクセス機能がスローする例外)を無視するようにした実装例を掲載する。

✧ ビジネスロジックの実装例

```
@Component
public class Sample03BLogic extends AbstractTransactionBLogic {

    @Inject
    @Named("csvFileQueryDAO")
    FileQueryDAO csvFileQueryDao;

    @Override
    public int doMain(BLogicParam param) {

        // Collector の生成
        Collector<Sample03Bean> collector = new FileCollector< Sample03Bean >(
                this.csvFileQueryDao, "inputFile/input_Sample03.csv", Sample03Bean.class);

        try {
            Sample03Bean inputData = null;
            while (collector.hasNext()) {
                try {
                    // データの取得
                    inputData = collector.next();
                } catch (FileException e) {
                    continue;
                }
                // DB の更新など、取得データに対する処理を記述する
            }
        } catch (Exception e) {
            // 例外処理
        } finally {
            // Collector のクローズ
            CollectorUtility.closeQuietly(collector);
        }
        return 0;
    }
}
```

データ取得部分を try-catch 文で囲む

ファイル読み込みに関する例外が発生した際は、例外はここでキャッチされ、以降の処理をスキップし、次のループ処理へと移行する。

必ず処理の最後にコレクタをクローズすること

➢ Collector クラスのコンストラクタについて

DaoCollector と FileCollector が用意するコンストラクタと、コンストラクタに使用される引数の一覧を掲載する。

➢ コンストラクタで設定できる内容について

実装例で使用した基本的なコンストラクタの他に、引数を与えることにより、以下の項目を設定することが可能である。

✧ TERASOLUNA Batch framework for Java が提供する 1:N マッピング機能の使用有無(DB のみ) (※1)

✧ キューサイズ

✧ 拡張例外ハンドラクラス(※2)

※1. MyBatis3 における select タグの resultOrdered 属性の値が true である場合と同等である。
1:N マッピングの処理を行う場合には、以下のページの collection の章を参照すること。
(http://mybatis.github.io/mybatis-3/ja/sqlmap-xml.html)

※2. 拡張例外ハンドラクラスに関しては、後述の拡張ポイントの項目を参照すること。

➢ Collector のコンストラクター一覧

先ほどの番号と合わせて以下にコンストラクタを列挙し、概要を掲載する。
引数についての詳細は、次ページのコンストラクタ引数一覧を参照すること。

✧ DaoCollector のコンストラクター一覧

コンストラクタ	概要
DaoCollector<P>(Object, String, Object)	実装例で掲載した基本となるコンストラクタ これら３つの引数は必須である。
DaoCollector<P>(Object, String, Object, boolean)	基本となるコンストラクタ及び、 1:N マッピング使用の有無を設定する。
DaoCollector<P>(Object, String, Object, int)	基本となるコンストラクタ及び、 キューサイズを設定する。
DaoCollector<P>(Object, String, Object, CollectorExceptionHandler)	基本となるコンストラクタ及び、 拡張例外ハンドラクラスを設定する。
DaoCollector<P>(Object, String, Object, int, boolean, CollectorExceptionHandler)	基本となるコンストラクタ及び、 1:N マッピング使用の有無、 キューサイズ、 拡張例外ハンドラクラスを設定する。

✧ FileCollector のコンストラクター一覧

コンストラクタ	概要
FileCollector<P>(FileQueryDAO, String, Class<P>)	実装例で掲載した基本となるコンストラクタ これら３つの引数は必須である。
FileCollector<P>(FileQueryDAO, String, Class<P>, CollectorExceptionHandler)	基本となるコンストラクタ及び、 拡張例外ハンドラクラスを設定する。
FileCollector<P>(FileQueryDAO, String, Class<P>, int, CollectorExceptionHandler)	基本となるコンストラクタ及び、 キューサイズ、 拡張例外ハンドラクラスを設定する。

➢ コンストラクタ引数一覧

前ページで列挙したコンストラクタで使用される引数の一覧を掲載する。

✧ DaoCollector のコンストラクタに渡される引数

引数	解説	デフォルト値	省略
Object	DB にアクセスするための DAO のインスタンス	―	不可
String	データを取得するための DAO のメソッド名	―	不可
Object	SQL にバインドされる値を格納したオブジェクト、バインドする値が存在しない場合は省略せず、null を渡すこと。	―	不可
int	キューサイズ、0 以下の値は無視される。基本的に変更不要(※3)	20	可
CollectorExceptionHandler	拡張例外ハンドラクラス	null	可
boolean	MyBatis の 1:N マッピング使用時は true を渡す。 true にすることにより、メモリの肥大化を最小限に抑えることができる。	false	可

✧ FileCollector のコンストラクタに渡される引数

引数	解説	デフォルト値	省略
FileQueryDAO	ファイルにアクセスするための DAO	―	不可
String	読み込むファイル名	―	不可
Class<P>	ファイル行オブジェクトクラス	―	不可
int	キューサイズ、0 以下の値は無視される。基本的に変更不要(※3)	20	可
CollectorExceptionHandler	拡張例外ハンドラクラス	null	可

※3. キューサイズの変更方針

基本的にはデフォルトの 20 から変更する必要はない。

デフォルト値で以下のような問題が発生する場合のみ変更すること。

✧ キューオブジェクトのサイズが大きすぎてヒープ領域を圧迫するような場合。
デフォルトの 20 からキューサイズを減少させる。

✧ キューサイズが小さすぎて、性能が低下している場合は、デフォルトの 20 からキューサイズを増加させる。
(ただし、キューサイズと性能は比例するものではないので、増やせば性能が向上するというものではない。自環境で十分な性能を発揮できる値を探し、設定すること)

◆ 拡張ポイント

➢ 拡張例外ハンドラクラスを独自実装する方法

デフォルトでは Collector#next メソッドを実行した際にビジネスロジックに例外がスローされるが、CollectorExceptionHandler インタフェースを実装した拡張入力チェックエラーハンドラクラスを独自実装することにより、エラー情報をもとにしたログ出力処理や、例外発生後の入力データ取得機能(コレクタ)の挙動の制御ができる。

➢ 拡張例外ハンドラクラスの実装

ここでは、以下の仕様を満たす拡張例外ハンドラクラスの実装例を掲載する。

【仕様】

① ログレベル ERROR で発生した例外情報をコンソールに残す。

② 例外発生後も処理は継続する。

✧ 拡張例外ハンドラクラスの実装例

```
public class CustomCollectorExceptionHandler implements CollectorExceptionHandler {

    private static Logger log = LoggerFactory.getLogger(CustomCollectorExceptionHandler.class);

    @Override
    public CollectorExceptionHandlerStatus handleException(
            DataValueObject dataValueObject) {

        // ログ出力
        if(log.isErrorEnabled()){
            log.error("以下の例外が発生しました。" , dataValueObject.getThrowable());
        }

        // CollectorExceptionHandlerStatus の設定
        return CollectorExceptionHandlerStatus.SKIP;
    }
}
```

handleException メソッドの実装を行う。

CollectorExceptionHandler インタフェースの実装を行う。

このメソッドは引数にエラー情報が格納された DataValueObject を持っている。

DataValueObject が持つエラー情報を利用したログ出力

ステータス:CollectorExceptionHandlerStatus.SKIP を返却することにより、後続の処理を継続する。(ステータスについては下表を参照)

この例では、ステータスとして「CollectorExceptionHandlerStatus.SKIP」を返却しているため、データ取得時に例外が発生した場合でも、次のデータを読み込む。ビジネスロジックで Collector#next メソッドを実行した際、例外が発生したデータは読み込まれない。

handleException メソッドが返却するステータスによって、例外発生後の Collector の挙動が制御される。

✧ ステータス:CollectorExceptionHandlerStatus 一覧表

CollectorExceptionHandlerStatus	挙動
SKIP	発生した例外はスローせず、 後続の処理を**継続**する(※1)。 (上の実装例にて使用)
END	発生した例外をスローせずに、 後続の処理を**停止**する。
THROW	発生した例外をそのままスローする。 (Collector#next メソッドを実行した際にビジネスロジックに例外がスローされる。拡張例外ハンドラクラスを独自実装しない場合の動作と同じ)

(※1)データベースからデータを取得する場合、コーディングポイントの例外処理で述べたように SQL 実行時エラー発生後に後続の処理を継続することはできないため、本拡張例外ハンドラクラスにてステータス[CollectorExceptionHandlerStatus.SKIP]を返却しても後続の処理を継続できないことに注意する。

➢ ビジネスロジックの実装

データ取得時に例外が発生した際に拡張例外ハンドラクラスの handleException メソッドが呼び出されるようにするためには、ビジネスロジック中で Collector のインスタンス生成時に拡張例外ハンドラクラスを渡しておく必要がある。

✧ ビジネスロジック実装例

```
@Component
public class Sample04BLogic extends AbstractTransactionBLogic {

    @Inject
    @Named("csvFileQueryDAO")
    FileQueryDAO csvFileQueryDao;

public int doMain(BLogicParam param) {
    // FileCollector の生成
    Collector<Sample04Bean> collector = new FileCollector<Sample04Bean>(
        this.csvFileQueryDAO, "inputFile/input_Sample04.csv",
        Sample04Bean.class, new CustomCollectorExceptionHandler());

    try {
        Sample04Bean inputData = null;
        while (collector.hasNext()) {
            // データの取得
            inputData = collector.next();
            // DB の更新など、取得データに対する処理を記述する
        }
    } catch (Exception e) {
        // 例外処理
    } finally {
        // Collector のクローズ
        CollectorUtility.closeQuietly(collector);
    }
        return 0;
}
```

コレクタ生成時に、先ほどの実装例で作成した拡張例外ハンドラクラスを渡す。

データ取得部分を try-catch 文で囲む必要がなくなる。

データ取得以外で発生した例外処理はここで行う。(システム例外など)

必ず処理の最後にコレクタをクローズすること

➢ Collector クラスを独自実装する方法

本機能が提供する AbstractCollector クラスの拡張クラスを作成することによって Collector クラスを独自実装することができる。

実装方法については、本機能が提供する FileCollector の実装を参考にすること。

■ リファレンス

◆ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.collector.Collector	Iterator インタフェースなどを拡張した Collector インタフェースクラス 現在の要素と前後の要素にアクセスするためのメソッドを宣言している。
2	jp.terasoluna.fw.collector.AbstractCollector	Collecotr インタフェースを実装した抽象クラス キューを保持するフィールドなどを持たせ、入力データ取得機能のメイン部分の処理をコーディングしている。
3	jp.terasoluna.fw.collector.db.DaoCollector	DB 用の AbstractCollector 拡張クラス DB からデータを取得する際に使用する。
4	jp.terasoluna.fw.collector.file.FileCollector	ファイル用の AbstractCollector 拡張クラス ファイルからデータを取得する際に使用する。
5	jp.terasoluna.fw.collector.db.QueueingResultHandler	ResultHandler インタフェースの拡張インタフェースクラス DB からデータを 1 件ずつ処理するためのいくつかのメソッドの宣言を行っている。
6	jp.terasoluna.fw.collector.db.QueueingResultHandlerImpl	QueueingResultHandler 実装クラス 取得したデータをキューに詰める部分などを実装したクラス。
7	jp.terasoluna.fw.collector.db.Queueing1NRelationResultHandlerImpl	QueueingResultHandler 実装クラス MyBatis の 1:N マッピングを利用する場合に使用する QueueingResultHandlerImpl の拡張クラスでもある。
8	jp.terasoluna.fw.collector.vo.DataValueObject	キューに詰められる ValueObject、取得したデータや取得時に発生した例外などが格納される。
9	jp.terasoluna.fw.collector.CollectorThreadFactory	データの取得を行う別スレッドを生成するためのスレッドファクトリクラス

◆ 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』

◆ 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

➢ **DB の更新処理を実施した後にデータを取得する際の注意事項**

コレクタによるデータの取得は、ビジネスロジックとは別のトランザクションで実行される。よって、DB の更新処理を実施し、コミットする前にコレクタを用いて、更新処理したデータを取得しようとした場合に、更新処理後のデータが取得できないので注意する必要がある。

例えば以下のような実装をした場合、パスワード「password」を DB に登録しても、コレクタで「豊洲」、「password」を取得できない。

➢ DB の更新処理を実施した後にコレクタを用いてデータを取得する実装例

```
@Component
public class B000001BLogic extends AbstractTransactionBLogic {
    ～省略～
    public int doMain(BLogicParam param) {
        InsertUser insertUser = new InsertUser();
        insertUser.setUserName("豊洲");
        insertUser.setPassword("password");
        //User テーブルにユーザ名「豊洲」、パスワード「password」のデータを登録
        b000001Dao.insertUser(inputDto);
       //User テーブルからコレクタを用いて全件データを取得する
       Collector<UserBean> collector = (省略)
       try{
            UserBean bean = null;
            while(collector.hasNext()){
                //入力データを取得しても
                //ユーザ名「豊洲」、パスワード「password」が取得できない
                bean = collector.next();
                 ･･･省略
             }
       ･･･省略
```

DB の更新処理を実施し、コミットする前に更新処理したデータを取得したい場合にはコレクタを利用せずに DAO のメソッドを直接用いること。

ただし、大量データ取得時にはメモリ枯渇の恐れがあるため、ResultHandler をメソッドの引数に持つ DAO を直接呼び出すこと。その際は、DAO のメソッドの引数に渡す ResultHandler の実装クラスを独自に作成する必要がある。この ResultHandler 実装クラスの handleResult メソッドに、データベースから取得したデータ 1 件ごとに実行したい処理を実装する。

➢ ビジネスロジックの実装例

```
<!-- コンポーネントスキャン設定 -->
@Component
public class B000001BLogic extends AbstractTransactionBLogic {
	private static Logger log = LoggerFactory.getLogger(B000001BLogic.class);
	@Inject
	B000001Dao b000001Dao = null;

	@Inject
	 ResultHandler<SelectUser> userDataResultHandler;

	public int doMain(BLogicParam param) {
		// 更新処理を実施
		b000001Dao.insertUser(inputDto);
		// SQL にバインドする値
		Object bindParams = null;
		// 更新処理後のデータを 1 件ずつ取得して、処理を実施
		b000001Dao.selectUser(bindParams, userDataResultHandler);
		return 0;
	}
}
```

更新処理後のデータを 1 件ずつ取得して、処理を実施する場合には、コレクタではなく ResultHandler を引数に持つ DAO を直接用いる。

➢ ResultHandler 実装クラスの実装例

```
@Component
public class UserDataResultHandler implements ResultHandler<SelectUser> {

	@Inject
	 UserDataDao userDataDao= null;
	// 取得したデータ 1 件毎に handleResult メソッドが呼ばれ、1 件分の
	// データが格納されたオブジェクトが引数に渡される
	 public void handleResult(ResultContext<? extends SelectUser> context) {
		// ResultContext オブジェクトから 1 件分のデータを取得
		SelectUser su = context.getResultObject();
		// 1 件のデータを処理するコードを記述
		～～省略～～
		// 更新処理を実施
		userDataDao.updateUser(su);
	}
}
```

※ResultHandler 内の DB 処理で例外を handleResult メソッド内で try-catch しない場合、DAO のメソッドから MyBatisSystemException がスローされる。ビジネスロジック側でこの例外をハンドリングしたい場合は、MyBatisSystemException を catch すること。

✧ パラメータで渡される ResultContext クラスのメソッドについて

	メソッド名	概要
1	getResultObject	select 要素の resultType 属性で指定した Java クラスのオブジェクトを取得するためのメソッド。
2	getResultCount	ResultHandler#handleResult メソッドの呼び出し回数を取得するためのメソッド。
3	stop	以降のレコードに対する処理を中止するように MyBatis 側に通知するためのメソッド。

※ResultConetxt には、isStopped というメソッドもあるが、これは MyBatis 側が使用するメソッドなので説明は割愛する。

➢ 大量データを検索する際の注意事項

大量のデータを返すようなクエリを記述する場合には、コレクタを使用するだけではなく、JDBC ドライバに対して適切な fetchSize を設定する必要がある。fetchSize 属性は、JDBC ドライバとデータベース間の通信において、一度の通信で取得するデータの件数を設定するパラメータである。

fetchSize 属性を省略した場合は、JDBC ドライバのデフォルト値が使用されるため、デフォルト値が全件取得する JDBC ドライバの場合、メモリの枯渇の原因になる可能性があるので、 注意が必要となる。

TERASOLUNA Batch3.6.x でサポートする MyBatis3.3.0 以降では、fetchSize の設定方法は 2 通りある。

- select タグの fetchSize 属性に SQL 単位の fetchSize を設定する
- MyBatis3 設定ファイルの defaultFetchSize 項目にデフォルトの fetchSize を設定する

設定についての詳細は、ガイドラインの「5.2.2.3.1. fetchSize の設定」(http://terasolunaorg.github.io/guideline/5.1.0.RELEASE/ja/ArchitectureInDetail/DataAccessMyBatis3.html#fetchsize)を参照のこと。

※例えば PostgreSQL JDBC ドライバのデフォルトは、一度の通信で検索対象のデータが全件取得される。数十件程度の件数であれば問題にならないが、数万件以上の大量データを検索する場合にはヒープメモリを圧迫してしまう可能性がある。