AL-041　入力データ取得機能(コレクタ)

■ 概要

◆ 機能全体像

- 入力データ取得機能の機能説明書では、機能を以下のように細分化し、それぞれの機能説明書で解説するものとする。
  - 入力データ取得機能(本機能説明書)
  - コントロールブレイク機能
  - 入力チェック機能

  入力データ取得機能の説明書では、入力データ取得機能の基本的な使用方法である、入力データの取得部分についての解説を行う。

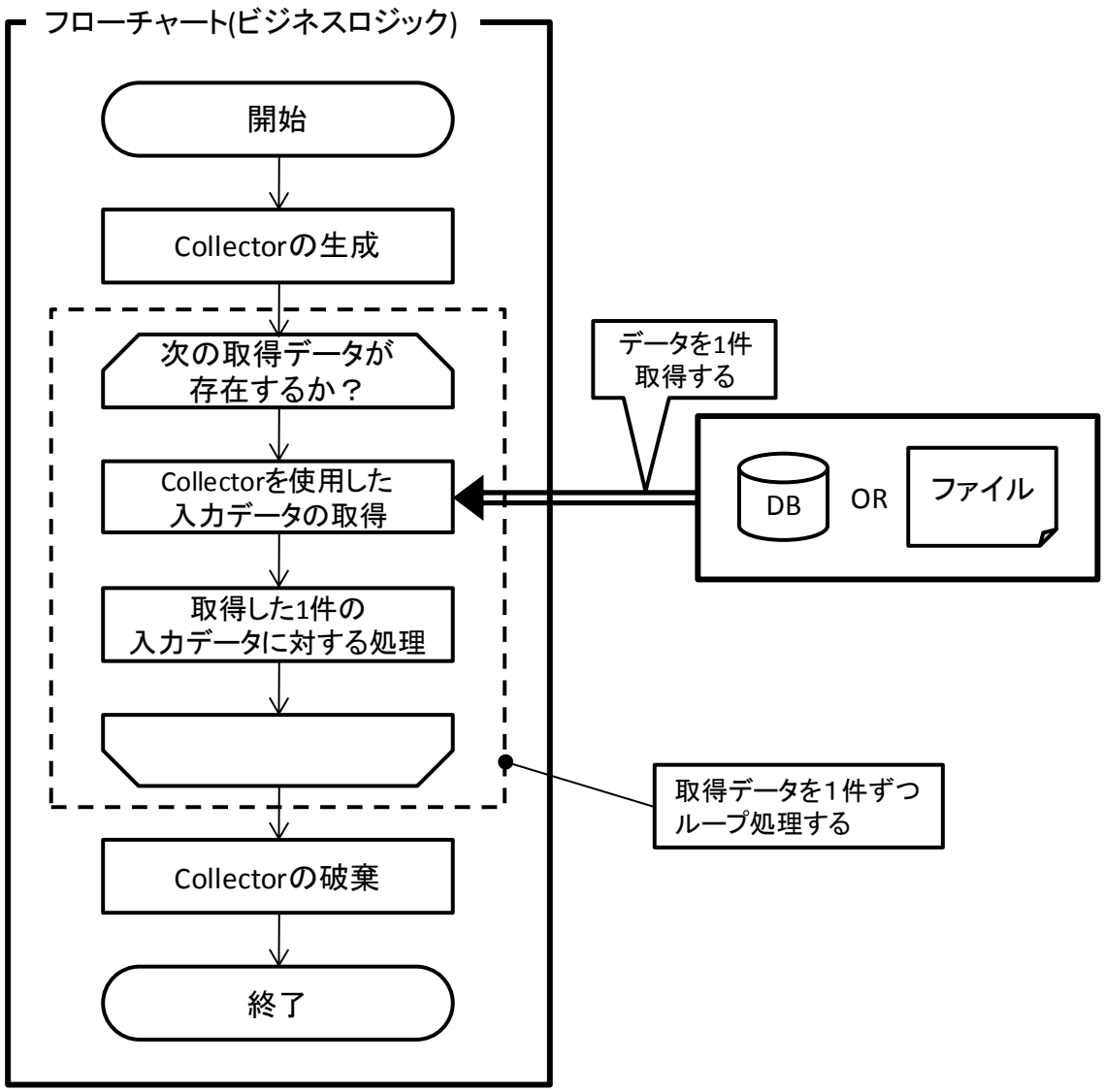
◆ 機能概要

- 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 からのデータ取得時に QueryRowHandleDAO を使用してデータを 1 件ずつ取得する。
- QueryDAO を使用してデータを一括で取得する場合と比べ、ヒープメモリの消費量を抑制する事ができる。
- 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事により、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事ができる。
    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事ができる。

◆ コーディングポイント

【コーディングポイントの構成】

- ビジネスロジックの実装例
  - データベースからデータを取得する際のビジネスロジックの実装例
  - CSV ファイルからデータを取得する際のビジネスロジックの実装例
    (データ取得時の例外発生時に、処理を停止するパターン)
  - CSV ファイルからデータを取得する際のビジネスロジックの実装例
    (データ取得時の例外発生時に、例外を無視して処理を継続するパターン)
- 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 のコンストラクター覧
  - コンストラクタ引数一覧

- ビジネスロジックの実装例
  以下に入力データ取得機能を利用して、データを取得する際の実装例を掲載する
  - データベースからデータを取得するビジネスロジックの実装例
    (TERASOLUNA Batch Framework for Java ver 3.x の場合)

```
@Component
public class Sample01BLogic extends AbstractTransactionBLogic {

    @Autowired
    protected QueryRowHandleDAO queryRowHandleDAO;

    @Override
    public int doMain(BLogicParam param) {

        // Collector の生成
        Collector<Sample01Bean> collector = new DBCollector<Sample01Bean>(
                this.queryRowHandleDAO, "Sample.selectData01", 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;
    }
}
```

DB からのデータ取得時は
QueryRowHandleDAO 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。
第一引数:QueryRowHandleDAO
第二引数:SQLID(文字列)
第三引数:SQL にバインドされるパラメータ

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事

- CSV ファイルからデータを取得するビジネスロジックの実装例
  (データ取得時の例外発生時に、処理を停止するパターン)
  (TERASOLUNA Batch Framework for Java ver 3.x の場合)

```
@Component
public class Sample02BLogic extends AbstractTransactionBLogic {

    @Autowired
    @Qualifier(value = "csvFileQueryDAO")
    protected 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。
第一引数: FileQueryDAO
第二引数:読み込むファイル名
第三引数: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。

このパターンでは、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事

ファイルの読み込みでは、読み込む際にフォーマットエラー等の例外が発生した場合に、例外を無視して処理を継続させる事が可能である。
次ページにデータ取得時の例外発生時に、例外を無視して処理を継続するパターンでの実装例を掲載する。

- CSV ファイルからデータを取得するビジネスロジックの実装例
  (データ取得時の例外発生時に、例外を無視して処理を継続するパターン)
  (TERASOLUNA Batch Framework for Java ver 3.x の場合)

```
@Component
public class Sample03BLogic extends AbstractTransactionBLogic {

    @Autowired
    @Qualifier(value = "csvFileQueryDAO")
    protected 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事

  - このように実装することで、データ取得時に FileException(ファイルアクセスで発生する例外のラップクラス)が発生した場合に、その行を無視して処理を継続させる事が可能である。

- Collector クラスのコンストラクタについて
  DBCollector 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、引数を与える事により、以下の項目を設定する事が可能である。
    - iBATIS の groupBy 属性使用の有無(DB のみ)(※１)
    - キューサイズ
    - 拡張例外ハンドラクラス(※２)

    ※1. iBATIS の groupBy 属性を使用する事によって、1:N 関係にあるテーブルの内容を、１つのクエリーで取得する事が出来る。
    詳細は iBATIS の機能説明書 P42 の「N+1 Selects を回避する」の項目を参照の事。(*http://ibatis.apache.org/docs/java/pdf/iBATIS-SqlMaps-2_ja.pdf*)

    ※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事。
  - Collector のコンストラクター覧
    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事。
    - DBCollector のコンストラクター覧

コンストラクタ	概要
DBCollector<P>(QueryRowHandleDAO, 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。
DBCollector<P>(QueryRowHandleDAO, 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。
DBCollector<P>(QueryRowHandleDAO, 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。
DBCollector<P>(QueryRowHandleDAO, 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。
DBCollector<P>(QueryRowHandleDAO, 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。

✧ DBCollector 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
QueryRowHandleDAO	DB にアクセスするための DAO	－	不可
String	SqlMap で定義した SQLID	－	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 を渡す事。	－	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	iBATIS の 1:N マッピング使用時は true を渡す。 true にする事により、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事。

✧ 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、設定する事）

◆ 拡張ポイント

- 独自実装した拡張例外ハンドリングを行う方法(ファイルからのデータ取得のみ)
  拡張例外ハンドラクラスを独自実装する事により、ログの出力方法・ログレベルを変更や、処理の継続・中断といった挙動の制御が可能になる。
  - ➢ 拡張例外ハンドラクラスの実装例
    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。

```
public class CustomCollectorExceptionHandler implements CollectorExceptionHandler {

    private Log log = LogFactory.getLog(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以降の処理を継続する。(ステータスについては下表を参照)

  - ➢ 拡張例外ハンドラクラスが返却するステータスについて
    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、 以降の処理を**継続**する。 (上の実装例にて使用)
END	発生した例外をスローせずに、 以降の処理を**停止**する。
THROW	発生した例外をそのままスローする

  - ➢ 拡張例外ハンドラクラスを使用する場合のビジネスロジック実装例
    ビジネスロジック中で Collector のインスタンスを生成するタイミングでコンストラクタを利用して、拡張例外ハンドラクラスを渡しておく必要がある。

ここでは FileCollector インスタンスの生成時のコーディングのみ記載し、ビジネスロジック全体像については掲載しない。

✧ ビジネスロジック実装例(FileCollector 生成部分のみ抜粋)

```
(略)

// FileCollector の生成
Collector<Sample01Bean> collector = new FileCollector<Sample01Bean>(
      this.csvFileQueryDAO, "inputFile/input_Sample1.csv",
      SampleFileLineObject.class, new CustomCollectorExceptionHandler());

(略)
```

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してやる。

以上のように FileCollector のインスタンスを生成する事により、Collector.getNext()によるデータ取得時に例外が発生した場合、自動的に拡張例外ハンドラクラスの handleException メソッドが呼び出され、独自実装された例外ハンドリング処理を行う。

- ➢ 拡張例外ハンドリングについての補足事項
  - ✧ DBCollector での本機能の使用について。
    DB からのデータの取得時にも本機能を使用する事は可能だが、ファイルからのデータの取得の場合と異なり、拡張例外ハンドラクラスにて、ステータス[CollectorExceptionHandlerStatuscollector.SKIP]を返却しても、例外発生後に Collector の処理を継続させる事は不可能である。

- ● 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事によって Collector クラスを独自実装する事が出来る。
  - ➢ 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事。

■ 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.DBCollector	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.QueueingDataRowHandler	DataRowHandler 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.QueueingDataRowHandlerImpl	QueueingDataRowHandler 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.Queueing1NRelationDataRowHandlerImpl	QueueingDataRowHandler 実装クラス iBATIS の 1:N マッピングを利用する場合に使用する QueueingDataRowHandlerImpl 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

- 『BB-01 データベースアクセス機能』
- 『BC-01 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
        updateDAO.execute("b000001.insertUser", insertUser);
       //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 QueryDAO もしくは QueryRowHandleDAO を用いること。特に大量データ取得時には QueryRowHandleDAO を用いること。

  実装するには、ビジネスロジック内で queryRowHandleDAO を利用する。また DataRowHandle インタフェースを実装したクラスで 1 件ずつ処理を行う。

➢ データソース Bean 定義ファイルの設定例

```
<!-- 照会系の QueryRowHandleDAO 定義 -->
<bean id="queryRowHandleDAO"
      class="jp.terasoluna.fw.dao.ibatis.QueryRowHandleDAOiBatisImpl">
      <property name="sqlMapClient" ref="sqlMapConfig"/>
</bean>
```

➢ ジョブ Bean 定義ファイルの設定例

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

➢ 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 Log log = LogFactory.getLog(B000001BLogic.class);
      @Autowired
      protected UpdateDAO updateDAO = null;
      @Autowired
      protected QueryRowHandleDAO queryRowHandleDAO = null;

      public int doMain(BLogicParam param) {
            ～省略～
            // 更新処理を実施
            updateDAO.execute("b000001.insertUser", insertUser);

            // SQL-ID
            String sqlID = "b000001.selectUser";
            // 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
            queryRowHandleDAO.executeWithRowHandler(sqlID, bindParams,
                                          userDataRowHandler);
            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 queryRowHandleDAO を用いる

➢ RowHandler 実装クラスの実装例

```
@Component
public class UserDataRowHandler implements DataRowHandler {
      @Autowired
       protected UpdateDAO updateDAO = null;
       // 1 件毎に handleRow メソッドが呼ばれ、引数に 1 件分の
       //データが格納されたオブジェクトが渡される
       public void handleRow(Object arg) {
             if (arg instanceof SelectUser) {
                   SelectUser su = (SelectUser) arg;
                   // 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
                   updateDAO.execute("b000001.updateUser", su);
              }
      }
}
```

- 大量データを検索する際の注意事項

  iBATIS マッピング定義ファイルの<statement> 要素、<select> 要素、<procedure> 要素にて大量データを返すようなクエリを記述する場合には、fetchSize 属性に適切な値を設定しておくこと。fetchSize 属性には 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。fetchSize 属性を省略した場合は各 JDBC ドライバのデフォルト値が利用される。

➢ SQL 定義ファイルの記述例

```
(略)

<select id=”sq0001” resultClass=”jp.terasoluna.xxx.bean.SampleXXXBean” fetchSize=”50”>
      SELECT ID,FAMILYNAME,FIRSTNAME,AGE FROM USER
</select>

(略)
```

大量データ取得時には、fetchSize を明示的に設定しておく。

※例えば PostgreSQL JDBC ドライバ(postgresql-8.3-604.jdbc3.jar にて確認) 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。