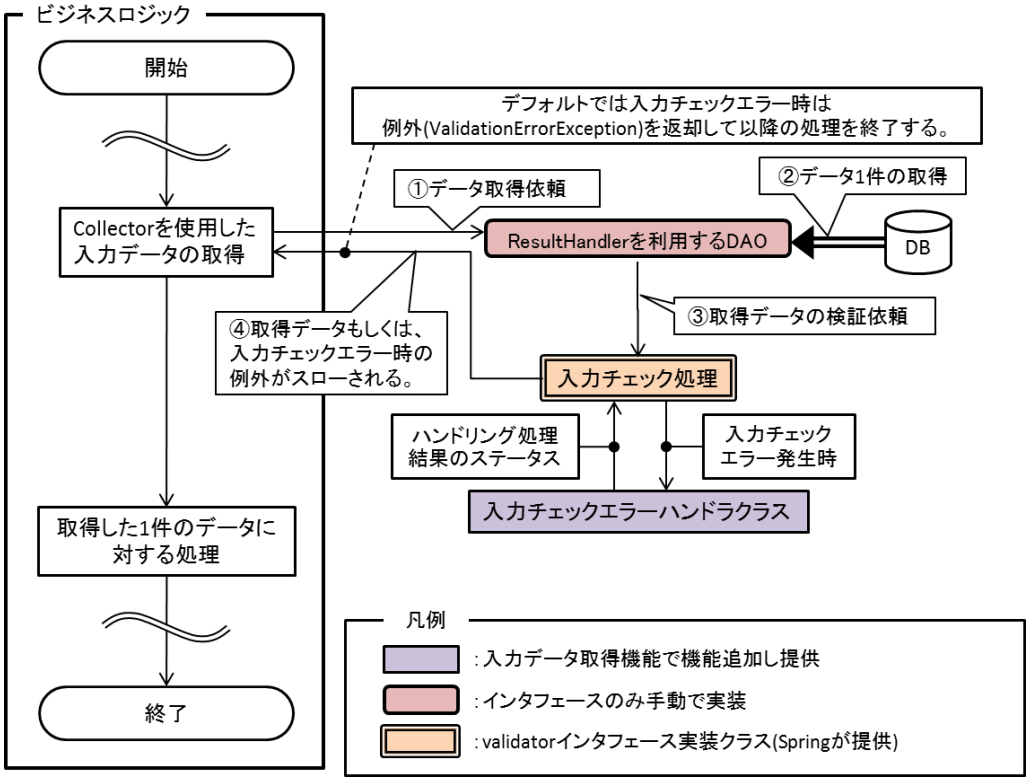
AL-043　入力チェック機能

■ 概要

◆ 機能概要

- 「AL-041 入力データ取得機能」を使用した際に、DB やファイルから取得したデータ 1 件毎に入力チェックを行う機能を提供する。
- 入力チェックは、DB やファイルからデータを取得するタイミングで行われる。
- アノテーション(Bean Validation)を利用した入力チェック機能を利用する。

◆ 概念図



◆ 解説

① Collector は、ResultHandler をメソッドの引数に持つ DAO(以降、入力チェック機能の説明に限り、単に DAO と呼ぶ)にデータの取得を依頼する。
② DAO は DB からデータを 1 件取得する。
③ DAO は取得したデータを返却する前に、validator インタフェース実装クラスに入力チェック処理を依頼する。
④ validator インタフェース実装クラスは入力チェックの結果に応じて、処理を振り分ける。
- 入力チェックエラーなしの場合…取得データをビジネスロジックに返却する。
- 入力チェックエラーありの場合…入力チェックエラーハンドラクラスによって入力チェックエラー時の例外「ValidationErrorException」がビジネスロジックに返却される。
  この時、独自に作成した拡張入力チェックエラーハンドラクラスを使用することによって、例外「ValidationErrorException」をスローすることなく以降の処理を継続させることも可能である。拡張入力チェックエラーハンドラクラスを作成する場合は、拡張ポイントの項目を参照すること。

◆ コーディングポイント

【コーディングポイントの構成】

- 入力チェックルールの設定例
  - アノテーションを利用した入力チェックルールの設定例
- 入力チェックを行う場合のビジネスロジックの実装例
  - ビジネスロジックの実装例(DB からのデータ取得)
  - ビジネスロジックの実装例(ファイルからのデータ取得)
- 本機能が提供する、入力チェックエラーハンドラクラスについて
- 入力チェック対応 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
  - 入力チェック対応 Collector クラスのコンストラクタ一覧
  - コンストラクタ引数一覧

- 入力チェックルールの設定例
  - 検証アノテーションを利用した入力チェックルールの設定例

入力チェック対象の DTO クラスのプロパティに対し、入力チェックルールを定義する検証アノテーションを付与することで、入力チェックを行うことができる。
以下に、入力チェックルールを定義する検証アノテーションを付与した DTO クラスの実装例を掲載する。

```
public class UserDto {

    @NotNull
    @Size(min=1, max=20)
    private String name;

    @NotNull
    @Min(0)
    @Max(200)
    private Integer age;

    // getter, setter が必要
}
```

上記の実装例では、次のような入力チェックを行うことができる。
- ✓ 名前(name)が null ではなく、1 文字以上、20 文字以下
- ✓ 年齢(age)が null ではなく、0 以上、200 以下

  - 検証アノテーション一覧

以下に、フレームワークが提供している検証アノテーション一覧を掲載する。

検証アノテーション	用途	使用例
@NotNull	対象のフィールドが、null でないことを検証する。	@NotNull private String id;
@Null	対象のフィールドが、null であることを検証する。	@Null private String id;
@Pattern	対象のフィールドが正規表現にマッチするかどうかを検証する。	@Pattern(regexp="[0-9]+") private String tel;
@Min	値が、最小値以上であるかどうかを検証する。	@Min(1) private int price;
@Max	値が、最大値以下であるかどうかを検証する。	@Max(100) private int age;
@DecimalMin	Decimal 型の値が、最小値以上であるかどうかを検証する。	@DecimalMin("0.0") private BigDecimal price;

@DecimalMax	Decimal 型の値が、最大値以下であるかどうかを検証する。	@DecimalMax(“9999.99”) private BigDecimal price;
@Size	Length が min と max の間のサイズか検証する。	@Size(min=4, max=64) private String password;
@Digits	値が指定された範囲内の数値であるかチェックする。 integer：整数部の最大桁数 fraction：小数部の最大桁数	@Digits(integer=6, fraction=2) private BigDecimal price;
@AssertTrue	対象のフィールドが true であることを検証する。	@AssertTrue private boolean checked;
@AssertFalse	対象のフィールドが false であることを検証する。	@AssertFalse private boolean checked;
@Future	実行サーバの現在時刻(ローカルのタイムゾーン)より未来時刻であるか検証する。	@Future private Date eventDate;
@Past	実行サーバの現在時刻(ローカルのタイムゾーン)より過去時刻であるか検証する。	@Past private Date eventDate;
@Valid	関連付けられているオブジェクトについて、再帰的に検証を行う。	@Valid private List<Employer> employers;
@CreditCardNumber	Luhn アルゴリズムでクレジットカード番号が妥当かどうか検証する。使用可能な番号かどうかをチェックするわけではない。「ignoreNonDigitCharacters=true」を指定することで、数字以外の文字を無視して検証することができる。	@CreditCardNumber private String cardNumber;
@Email	RFC2822 に準拠した Email アドレスかどうか検証する。	@Email private String email;
@URL	RFC2396 に準拠しているかどうかを検証する。	@URL private String url;
@NotBlank	トリムされた文字列長が 0 より大きいことを検証する	@NotBlank private String userId;
@NotEmpty	Null、または空文字(“”)でないことを検証する	@NotEmpty Private String password;

- 入力チェックを行う場合のビジネスロジックの実装例
  以下に DB、及びファイルからデータを取得する際に入力チェックを行う際の実装例を掲載する。
  使用する Collector クラスが、入力チェックを行わない場合と異なる点に注意する。
  - ビジネスロジックの実装例(DB からのデータ取得)

```
@Component
public class Sample01BLogic extends AbstractTransactionBLogic {
    private static final Logger logger = LoggerFactory.getLogger(Sample01BLogic.class);

    @Inject
    protected Sample01Dao sample01Dao;

    @Inject
    protected Validator validator;

    @Override
    public int doMain(BLogicParam param) {

        // Collector の生成
        Collector<Sample01Dto> collector = new DaoValidateCollector<Sample01Dto>(
                this.sample01Dao, "collectData01", null, validator);

        try {
            Sample01Dto inputData = null;
            while (collector.hasNext()) {
                // データの取得
                inputData = collector.next();

                // 取得データに対する処理(ここでは省略する)
            }
        } catch (ValidationErrorException e) {
            // 例外処理
            logger.warn("{}件目で検証エラーです。", e.getDataValueObject().getDataCount());
        } finally {
            // Collector のクローズ
            CollectorUtility.closeQuietly(collector);
        }
        return 0;
    }
}
```

Validator の DI を行う

DaoValidateCollector を生成する。
DaoCollector と異なり、第四引数に入力チェックを行う Validator クラスを渡している点に注意すること。

入力チェックの実行や、入力チェックエラーハンドラの実行は、非同期で事前に対象データが取得されたタイミングで行われ、next メソッドを実行した際に、次に取得する対象データの入力チェック結果がフィードバックされる。
Collector クラスのコンストラクタにて、(拡張)入力チェックエラーハンドラを渡していない場合、入力チェックエラー発生時に「ValidationErrorException」がスローされる。

「ValidationErrorException」を catch し、入力チェックエラー内容を出力する。

必ず finally ブロックを記述し、この中でコレクタをクローズすること。

➢ ビジネスロジックの実装例(ファイルからのデータ取得)

```
@Component
public class Sample02BLogic extends AbstractTransactionBLogic {
    private static final Logger logger = LoggerFactory.getLogger(Sample02BLogic.class);

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

    @Inject
    protected Validator validator;

    @Override
    public int doMain(BLogicParam param) {
        // Collector の生成
        Collector<Sample02Dto> collector = new FileValidateCollector<Sample02Dto>(
                this.csvFileQueryDao, "inputFile/SampleFile.csv", Sample02Dto.class, validator);

        try {
            Sample02Dto 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 (ValidationErrorException e) {
            // 例外処理
            logger.warn("{}件目で検証エラーです。", e.getDataValueObject().getDataCount());
        } finally {
            // Collector のクローズ
            CollectorUtility.closeQuietly(collector);
        }
        return 0;
    }
}
```

Validator の DI を行う

FileValidateCollector を生成する。
FileCollector と異なり、第四引数に入力チェックを行う Validator クラスを渡している点に注意すること。

入力チェックの実行や、入力チェックエラーハンドラの実行は、非同期で事前に対象データが取得されたタイミングで行われ、next メソッドを実行した際に、次に取得する対象データの入力チェック結果がフィードバックされる。
Collector クラスのコンストラクタにて、(拡張)入力チェックエラーハンドラを渡していない場合、入力チェックエラー発生時に「ValidationErrorException」がスローされる。

「ValidationErrorException」を catch し、入力チェックエラー内容を出力する。

必ず finally ブロックを記述し、この中でコレクタをクローズすること。

- 本機能が提供する、入力チェックエラーハンドラクラスについて

入力チェックエラーハンドラクラス	仕様
ExceptionValidationErrorHandler	デフォルトで使用される入力チェックエラーハンドラクラス。入力チェックエラーが発生した時点で例外をスローする。この入力チェックエラーハンドラは以下の場合に使用する。 ✓ 入力チェックエラー検出時に処理を異常終了させる場合 ✓ 入力チェックエラー例外を呼び出し元でハンドリングして、処理を継続(次のデータを処理)する場合

- 入力チェック対応 Collector クラスのコンストラクタについて
  DaoValidateCollector と FileValidateCollector 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。
    ① 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)
    ② キューサイズ
    ③ 拡張例外ハンドラクラス(※２)
    ④ 使用する入力チェックエラーハンドラクラス(※３)

    ※1. 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、『AL-041 入力データ取得機能』の機能説明書の拡張ポイントの項目を参照すること。
    ※3. デフォルトでは先に紹介した「ExceptionValidationErrorHandler」が使用される、独自にハンドラクラスを作成することも可能。
    ハンドラクラスを独自実装する場合は後述の拡張ポイントの項目を参照の事。

- 入力チェック対応 Collector クラスのコンストラクター一覧
  以下に入力チェック対応の Collector クラスの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、後述のコンストラクタ引数一覧を参照すること。
  - DaoValidateCollector のコンストラクター一覧

コンストラクタ	概要
DaoValidateCollector<P>(Object, String, Object, Validator)	実装例で掲載した基本となるコンストラクタ これら４つの引数は必須である。
DaoValidateCollector<P>(Object, String, Object, Validator, ValidationErrorHandler)	基本となるコンストラクタ及び、 使用する入力チェックエラー ハンドラクラスを設定する。
DaoValidateCollector<P>(Object, String, Object, boolean, Validator)	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。
DaoValidateCollector<P>(Object, String, Object, boolean, Validator, ValidationErrorHandler)	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、 使用する入力チェックエラーハンドラクラスを設定する。
DaoValidateCollector<P>(Object, String, Object, int, Validator)	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。
DaoValidateCollector<P>(Object, String, Object, int, Validator, ValidationErrorHandler)	基本となるコンストラクタ及び、 キューサイズ、 使用する入力チェックエラーハンドラクラスを使用する。
DaoValidateCollector<P>(Object, String, Object, int, CollectorExceptionHandler, Validator)	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。
DaoValidateCollector<P>(Object, String, Object, int, CollectorExceptionHandler, Validator, ValidationErrorHandler)	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、 使用する入力チェックエラーハンドラクラスを設定する。
DaoValidateCollector<P>(Object, String, Object, int, boolean, CollectorExceptionHandler, Validator)	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。
DaoValidateCollector<P>(Object, String, Object, int, boolean, CollectorExceptionHandler, Validator, ValidationErrorHandler)	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、 使用する入力チェックエラーハンドラクラスを設定する。

✧ FileValidateCollector のコンストラクタ一覧

コンストラクタ	概要
FileValidateCollector<P>(FileQueryDAO, String, Class<P>, Validator)	実装例で掲載した基本となるコンストラクタ これら４つの引数は必須である。
FileValidateCollector<P>(FileQueryDAO, String, Class<P>, Validator, ValidationErrorHandler)	基本となるコンストラクタ及び、 使用する入力チェックエラーハンドラクラスを設定する。
FileValidateCollector<P>(FileQueryDAO, String, Class<P>, CollectorExceptionHandler, Validator)	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。
FileValidateCollector<P>(FileQueryDAO, String, Class<P>, CollectorExceptionHandler, Validator, ValidationErrorHandler)	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、 使用する入力チェックエラーハンドラクラスを設定する。
FileValidateCollector<P>(FileQueryDAO, String, Class<P>, int, CollectorExceptionHandler, Validator)	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。
FileValidateCollector<P>(FileQueryDAO, String, Class<P>, int, CollectorExceptionHandler, Validator, ValidationErrorHandler)	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、 使用する入力チェックエラー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を以下に列挙する。
  入力データ取得機能と比較し、差分となる新規要素については**太字**で掲載する
  - ✧ DaoValidateCollector のコンストラクタで渡される引数

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	ResultHandler クラスを引数にもつ 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。	20	可
CollectorExceptionHandler	例外ハンドラクラス、	null	可
boolean	MyBatis3 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
Validator	**入力チェックを行う Validator。通常は Spring が提供する Validator を使用する。**	―	不可
ValidationErrorHandler	**入力チェックエラーハンドラクラス。**	**ExceptionValidationErrorHandler**	**可**

  - ✧ FileValidateCollector のコンストラクタで渡される引数

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。	20	可
CollectorExceptionHandler	例外ハンドラクラス、	null	可
Validator	**入力チェックを行う Validator。通常は Spring が提供する Validator 使用する。**	―	不可
ValidationErrorHandler	**入力チェックエラーハンドラクラス。**	**ExceptionValidationErrorHandler**	**可**

拡張ポイント

- 独自に検証アノテーションを実装する方法
  フレームワークが提供していない単項目チェックルールや相関項目チェックルールの検証アノテーションを追加したい場合は、以下の URL の「How to extend」を参考にすること。
  (http://terasolunaorg.github.io/guideline/5.0.0.RELEASE/ja/ArchitectureInDetail/Validation.html#how-to-extend)

- 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方法
  - ValidationErrorHandler インタフェースの実装クラスを作成することにより、拡張入力チェックエラーハンドラクラスを作成することが可能である。
  - 拡張入力チェックエラーハンドラクラスは、ExceptionValidationErrorHandler のように例外をスローする他、以降の処理を制御するステータス ValidateErrorStatus を返却することができる。
  - ValidateErrorStatus の一覧表
    入力チェックエラーハンドラクラスが返却するステータス一覧と、各ステータスが返却された際にビジネスロジック側でデータを取得する時の挙動について説明する。

ValidateErrorStatus	Collector の next メソッドの取得対象が入力チェックエラーデータの場合の挙動	Collector の getNext (getPrevious) メソッドの取得対象が入力チェックエラーデータの場合の挙動
SKIP	**エラーデータは取得せずに**、その後の正常なデータを取得する。その後の処理は継続する。	**エラーデータは取得せずに**、その後の正常なデータを取得する。
CONTINUE	**エラーデータを取得する。**その後の処理は継続する。	**エラーデータを取得する。**
END	エラーデータは取得せず、以降のデータも取得しない。 ※事前の hasNext による問合せに false を返す。	getNext は **null** を返却する(次のデータが存在しない、終端を意味する)。 getPrevious では参照できない。
なし (例外がスローされた場合)	例外がスローされる。ビジネスロジックで処理を停止しない限り、処理は継続する。	**エラーデータを取得する。**

※コントロールブレイク機能では、コントロールブレイク判定時に使用されるデータは後ブレイク判定の場合は getNext メソッド、前ブレイク判定の場合は getPrevious メソッドで前後のデータを取得し、ブレイク判定を行っている。コントロールブレイク判定時の比較対象のデータ getNext、getPrevious メソッドの返却値を意識すること。

- 以下に拡張入力チェックエラーハンドラクラスの実装例を掲載する。
  実装例では拡張入力チェックエラーハンドラクラスは以下の仕様で作成する。
  【仕様】
  ① 入力チェックエラー発生時にログレベル info でエラー発生を通知する。
  ② 入力チェックエラーが発生したデータは無視し、以降の処理を継続する。
- 拡張入力チェックエラーハンドラクラス実装例

```
public class CustomValidationErrorHandler implements ValidationErrorHandler {

    private static Logger logger =
        LoggerFactory.getLogger(CustomValidationErrorHandler.class);

    @Override
    public ValidateErrorStatus handleValidationError(
            DataValueObject dataValueObject, Errors errors) {

        // ログ出力
        if(logger.isInfoEnabled()){
            logger.info("入力チェックエラー発生");
        }

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

ValidationErrorHandler インタフェースを実装する。

handleValidationError メソッドの実装を行う

仕様①に従い、info レベルでエラーの発生を通知する。入力チェックエラーの内容を出力する例は、後述する。

仕様②に従い、SKIP を返却することにより、エラーが発生したデータを無視して、その後の処理を継続する。

➢ ビジネスロジックの実装例(DB)

```
@Component
public class Sample03BLogic extends AbstractTransactionBLogic {
    private static final Logger logger = LoggerFactory.getLogger(Sample03BLogic.class);

    @Inject
    protected Sample03Dao sample03Dao;

    @Inject
    private Validator validator;

    @Inject
    private CustomValidationErrorHandler handler;

    @Override
    public int doMain(BLogicParam param) {

        // Collector の生成
        Collector<Sample03Dto> collector = new DaoValidateCollector<Sample03Dto>(
                this.sample03Dao, " collectData06", null,
                validator, handler);

        try {
            Sample03Dto inputData = null;
            while (collector.hasNext()) {
                // データの取得
                inputData = collector.next();

            }
        } catch (ValidationErrorException e) {
            // 例外処理
            logger.warn("{}件目で検証エラー
        } finally {
            // Collector の破棄
            CollectorUtility.closeQuietly(collector);
        }
    return 0;
    }
}
```

Validator の DI を行う。

独自実装した拡張入力チェックエラーハンドラクラスの DI を行う。ハンドラクラスがスレッドアンセーフの場合は、new で生成すること。

コレクタ生成時に上で DI(もしくは new)した拡張入力チェックエラーハンドラクラスを渡しておく。

入力チェックの実行や、拡張入力チェックエラーハンドラの実行は、非同期で事前に対象データが取得されたタイミングで行われ、next メソッドを実行した際に、次に取得する対象データの入力チェック結果がフィードバックされる。拡張入力チェックエラーハンドラが SKIP や END を返す場合、next メソッドで取得できる件数自体が変わるため、入力チェック結果は、hasNext メソッドにも影響を及ぼす。例えば、SKIP や END の結果、next メソッドが参照できるデータが 1 つもなくなるケースにおいては、直前の hasNext メソッド呼び出し時に false を返す。

このように Collector インスタンス生成時にあらかじめ拡張入力チェックエラーハンドラクラスを渡すことにより、入力チェックエラー発生時にはこのハンドラクラスが使用されることになる。

- 入力チェックエラーの内容を確認する方法
  入力チェックエラーの内容は、入力チェックエラーハンドラクラスの handleValidationError メソッドに渡される Errors オブジェクトに格納されている FieldError オブジェクトから取得する。以下に、FieldError オブジェクトから入力チェックエラーの内容を取得する方法を掲載する。

  ➢ 入力チェックエラーメッセージのデフォルトのメッセージ定義
  入力チェックエラー時のデフォルトのメッセージは、ValidationMessages.properties に定義する。

```
javax.validation.constraints.Max.message={value}以下で入力してください。
```

「検証アノテーションクラスの FQCN + .message」のプロパティキーを利用する。

  ➢ 入力チェックエラーの内容を取得する方法

```
public class CustomValidationErrorHandler implements ValidationErrorHandler {

    private static Logger logger =
        LoggerFactory.getLogger(CustomValidationErrorHandler.class);

    @Override
    public ValidateErrorStatus handleValidationError(
            DataValueObject dataValueObject, Errors errors) {

        List<FieldError> fieldErrorsList = errors.getFieldErrors();

        for (FieldError fe : fieldErrorsList) {
            // FieldError オブジェクトからメッセージを取得し
            // 入力チェックエラーメッセージを出力
            logger.warn(fe.getDefaultMessage());
        }

    }
}
```

FieldError オブジェクトに、1 件分の入力チェックエラー内容が格納されている。例えば、1 レコード中に 3 件エラーがある場合は、3 つの入力チェックエラー内容が格納されている

  ➢ FieldError オブジェクトから取得できる入力チェックエラーの内容
  FieldError オブジェクトから入力チェックエラーの内容を取得するためのメソッドを以下に掲載する。

取得方法	返却される型	取得内容
getDefaultMessage	String	入力チェックルールに対応するメッセージを ValidationMessage.properties から取得する。上記の例にて、Max 検証ルール(例:100 以下)で入力チェックエラーが発生した場合は、「100 以下で入力してください。」を返却する。
getArguments	Object[]	入力チェックエラーメッセージを解決するために使用される値を返却する。
getRejectedValue	String	入力チェックエラーとなった対象の値を返却する。
getObjectName	String	入力チェック対象の DTO クラスのオブジェクト名を返却する。
getField	String	入力チェックエラーとなった対象のプロパティ名を返却する。
getCodes	String[]	入力チェックエラーのメッセージと対応付けるコードのリストを返却する。リストの各要素の例を以下に示す。配列等を使い、ネストしたクラスを入力チェックする場合は、要素が増える場合があるため、要素のインデックスを考慮すること。 [0] : ${検証ルール名}.${オブジェクト名}.${プロパティ名} [1] : ${検証ルール名}.${プロパティ名} [2] : ${検証ルール名}.${検証対象のプロパティの型} [3] : ${検証ルール名}
getCode	String	getCodes の最後の要素を返却する。

また、検証アノテーションの message 属性に、直接メッセージを指定することで、getDefaultMessage メソッドにて取得できるメッセージを変更することができる。
検証アノテーションで指定できるメッセージの形式は次の通りである

- ✧ 検証アノテーションに直接メッセージを設定する
- ✧ 検証アノテーションにプロパティキーを設定する

以下に、それぞれの設定例を掲載する。

✧ 検証アノテーションに直接メッセージを設定する

検証アノテーションに直接取得したいメッセージを取得すると、FieldError オブジェクトの getDefaultMessage メソッドでそのメッセージを取得できる。

```
public SampleBean {

    @NotNull(message = “名前は入力必須です。”)
    private String name;

    // setter, getter は必須
}
```

検証アノテーションの message 属性にメッセージを直接指定する。

name プロパティで入力チェックエラーが発生した場合、FieldError オブジェクトの getDefaultMessage メッセージの返却値として「名前は入力必須です。」が取得できる。

✧ 検証アノテーションにプロパティキーを設定する

プロパティファイルに定義した別のメッセージキーのメッセージを出力させることができる。プロパティファイルは、commonContext.xml に定義されている messageSource の basenames プロパティファイルでなければならない。

✓ プロパティファイルの設定例

```
sample.required=入力は必須です。
```

✓ 検証アノテーションの設定例

```
public SampleBean {

    @NotNull(message = “{sample.required}”)
    private String name;

    // setter, getter は必須
}
```

“{プロパティキー}"を指定する。

上記の設定例の場合、FieldError オブジェクトの getDefaultMessage メッセージの返却値として「入力は必須です。」が取得できる。

- 入力チェックエラーメッセージ内容を出力する方法

入力チェックエラーメッセージを取得する際、以下の方法を利用しても、メッセージを取得することができる。

① フレームワークが提供している MessageSource を利用して入力チェックエラーメッセージを取得する

② フレームワークが提供している MessageUtil を利用して入力チェックエラーメッセージを取得する

それぞれについて、以下に実装例を掲載する。

① フレームワークが提供している MessageSource を利用して入力チェックエラーメッセージを取得する

➢ 入力チェックエラーメッセージのデフォルトのメッセージ定義の設定例

入力チェックエラー時のデフォルトのメッセージを、src/main/resources 配下にある ValidationMessages.properties に定義する。

```
javax.validation.constraints.Max.message={0}は{value}以下で入力してください。
```

「検証アノテーションクラスの FQCN + .message」のプロパティキーを指定する。

➢ MessageSource を使って入力チェックエラーの内容を取得する方法

```
public class CustomValidationErrorHandler implements ValidationErrorHandler {

    private static Logger logger =
        LoggerFactory.getLogger(CustomValidationErrorHandler.class);
    @Inject
    MessageSource messageSource;

    @Override
    public ValidateErrorStatus handleValidationError(
            DataValueObject dataValueObject, Errors errors) {

        List<FieldError> fieldErrorsList = errors.getFieldErrors();

        for(FieldError fe : fieldErrorsList) {
            logger.warn(messageSource.getMessage(fe, Locale.getDefault()));
        }

    }
}
```

ValidationMessages.properties に定義されている「{0}は{value}以下で入力してください。」のメッセージの{0}にはプロパティ名、{value}には検証値の値が置換された入力チェックエラーメッセージが取得できる(@Size(min=1, max=10)のような場合では、{min}, {max}や{message}のようにアノテーションのフィールド名で、その検証値を取得できる)。kingakuというプロパティに@Max(100)の入力チェックをした場合、「kingaku は 100 以下で入力してください。」のメッセージが取得できる。

➢ 個別に名前を変更したい場合の設定例

src/main/resources 配下にある application-messages.properties の定義追加が必要となる。

✓ application-messages.properties の設定例

```
kingaku=金額
```

メッセージキーに置換対象の JavaBean のプロパティ名、メッセージに置換後の文字列を設定する

kingaku というプロパティ名が、「金額」という文字列に置換され、入力チェックエラーメッセージは「金額は 100 以下で入力してください。」となる。

② フレームワークが提供している MessageUtil を利用して入力チェックエラーメッセージを取得する

➢ 入力チェックエラーメッセージのデフォルトのメッセージ定義の設定例

```
NotEmpty={0}は入力必須です。
```

MessageUtil を利用して取得するメッセージを定義する。

➢ MessageUtil を使って入力チェックエラーの内容を取得する方法

```
public class CustomValidationErrorHandler implements ValidationErrorHandler {

    private static Logger logger =
        LoggerFactory.getLogger(CustomValidationErrorHandler.class);

    @Override
    public ValidateErrorStatus handleValidationError(
            DataValueObject dataValueObject, Errors errors) {
        List<FieldError> fieldErrorsList = errors.getFieldErrors();

        for(FieldError fe : fieldErrorsList) {
            logger.warn(MessageUtil.getMessage(fe.getCode(), fe.getField()));
        }
    }
}
```

name というプロパティに@NotEmpty を設定した場合、fe.getCode()では”NotEmpty”の文字列、getField では”name”の文字列が取得できる。
第一引数:取得したいメッセージのプロパティキー
第二引数以降:プレースホルダに代入したい文字列
上記の例では、プロパティキーが NotEmpty のメッセージを取得し、{0}に name を代入した「name は入力必須です。」という文字列を取得できる。
※MessageSource と違い、{0}はプロパティ名ではなく、MessageUtil#getMessage の第二引数が代入される。以降、{1}には、第三引数、{2}には第四引数・・・が代入される。

■ リファレンス

◆ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.collector.db.DaoValidateCollector	DaoCollector 拡張クラス DaoCollector を入力チェックに対応させている。
2	jp.terasoluna.fw.collector.file.FileValidateCollector	FileCollector 拡張クラス FileCollector を入力チェックに対応させている。
3	jp.terasoluna.fw.collector.validate.ValidationErrorHandler	入力チェックエラーハンドラインタフェース 入力チェックエラーが発生した際の処理を宣言している。
4	jp.terasoluna.fw.collector.validate.AbstractValidationErrorHandler	ValidationErrorHandler クラスを実装した抽象クラス コンストラクタによるログレベルの変更や、ログ出力用のメソッドなどの処理を定義している。
5	jp.terasoluna.fw.collector.validate.ExceptionValidationErrorHandler	AbstractValidationErrorHandler クラスの拡張クラス 入力チェックエラーが発生した場合は TRACE ログにエラーコードを出力し、例外(ValidationErrorException)をスローする(処理が途中で中断する)
6	jp.terasoluna.fw.collector.validate.ValidateErrorStatus	列挙型クラス 入力チェックエラーハンドラクラスはこの値によって、入力チェックエラー発生後の挙動を決定する。
7	jp.terasoluna.fw.collector.validate.ValidationErrorException	RuntimeException を拡張した入力チェックエラークラス 入力チェックエラー発生時にスローされる。

◆ 関連機能

- 『AL041 入力データ取得機能』

◆ 使用例

- **機能網羅サンプル**(terasoluna-batch-functionsample)
- **チュートリアル**(terasoluna-batch-tutorial)

◆ 注意事項

なし

◆ 備考

なし