



TERASOLUNA Batch Framework for Java

引用資料

第 3.2.0 版

NTT DATA

株式会社 NTTデータ

本ドキュメントを使用するにあたり、以下の規約に同意していただく必要があります。同意いただけない場合は、本ドキュメント及びその複製物の全てを直ちに消去又は破棄してください。

1. 本ドキュメントの著作権及びその他一切の権利は、NTT データあるいは NTT データに権利を許諾する第三者に帰属します。
2. 本ドキュメントの一部または全部を、自らが使用する目的において、複製、翻訳、翻案することができます。ただし本ページの規約全文、および NTT データの著作権表示を削除することはできません。
3. 本ドキュメントの一部または全部を、自らが使用する目的において改変したり、本ドキュメントを用いた二次的著作物を作成することができます。ただし、「TERASOLUNA Batch Framework for Java (機能説明書)」あるいは同等の表現を、作成したドキュメント及びその複製物に記載するものとします。
4. 前2項によって作成したドキュメント及びその複製物を、無償の場合に限り、第三者へ提供することができます。
5. NTT データの書面による承諾を得ることなく、本規約に定められる条件を超えて、本ドキュメント及びその複製物を使用したり、本規約上の権利の全部又は一部を第三者に譲渡したりすることはできません。
6. NTT データは、本ドキュメントの内容の正確性、使用目的への適合性の保証、使用結果についての的確性や信頼性の保証、及び瑕疵担保義務も含め、直接、間接に被ったいかなる損害に対しても一切の責任を負いません。
7. NTT データは、本ドキュメントが第三者の著作権、その他如何なる権利も侵害しないことを保証しません。また、著作権、その他の権利侵害を直接又は間接の原因としてなされる如何なる請求(第三者との間の紛争を理由になされる請求を含む。)に関しても、NTT データは一切の責任を負いません。

本ドキュメントで使用されている各社の会社名及びサービス名、商品名に関する登録商標および商標は、以下の通りです。

Terasoluna は、株式会社 NTT データの登録商標です。

その他の会社名、製品名は、各社の登録商標または商標です。

目次

- TERASOLUNA Batch Framework for Java 引用資料一覧

CB-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

BC-02 ファイル操作機能

CE-01 メッセージ管理機能

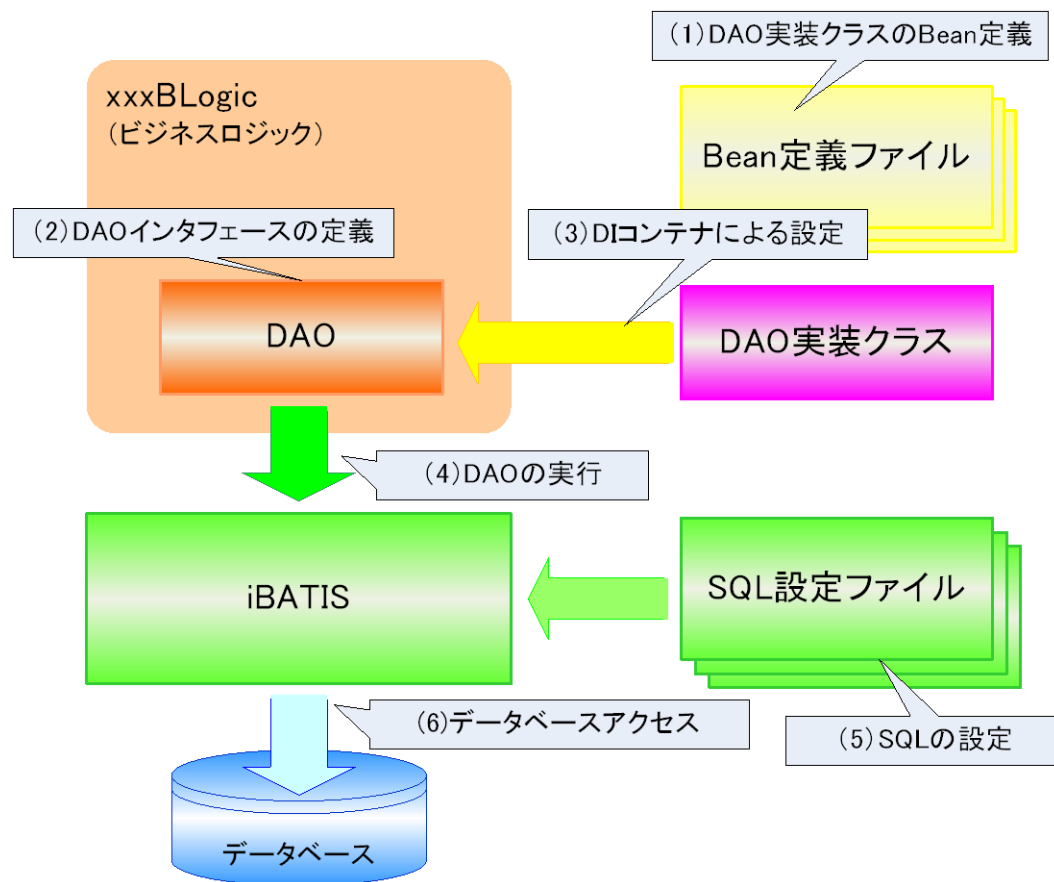
CB-01 データベースアクセス機能

■ 概要

◆ 機能概要

- データベースアクセスを簡易化する DAO を提供する。
- 以下の DAO インタフェースを提供し、JDBC API および RDBMS 製品や O/R マッピングツールに依存する処理を業務ロジックから隠蔽化する。
 - QueryDAO
データを検索する際に使用する。
 - UpdateDAO
データを挿入・更新・削除する際に使用する。
 - StoredProcedureDAO
ストアドプロシジャを発行する際に使用する。
 - QueryRowHandleDAO
大量データを検索し一件ずつ処理する際に使用する。
- DAO インタフェースのデフォルト実装として Spring + iBATIS 連携機能を利用した以下の DAO 実装クラスを提供する。
 - QueryDAOiBatisImpl
iBATIS に対してデータを検索する際に使用する。
 - UpdateDAOiBatisImpl
iBATIS に対してデータを挿入・更新・削除する際に使用する。
 - StoredProcedureDAOiBatisImpl
iBATIS に対してストアドプロシジャを発行する際に使用する。
 - QueryRowHandleDAOiBatisImpl
iBATIS に対して大量データを検索し一件ずつ処理する際に使用する。
- AOP を利用した宣言的トランザクション制御を行うため、業務ロジック実装者が、コネクションオブジェクトの受け渡しなどのトランザクションを考慮した処理を実装する必要がない。
- iBATIS の詳細な使用方法や設定方法などは、iBATIS のリファレンスを参照すること。
- SQL 文は、iBATIS の仕様にしたがって、設定ファイルにまとめて記述する。

◆ 概念図



◆ 解説

- (1) DAO 実装クラスのオブジェクトを Bean 定義ファイルに定義する。
- (2) ビジネスロジックには DAO を利用するために DAO インタフェースの属性およびその Setter を用意する。
- (3) DI コンテナによってビジネスロジックを生成する際、(1)で定義した DAO オブジェクトを属性に設定するために、データアクセスを行うビジネスロジックの Bean 定義に、(1)で定義した DAO 実装クラスを設定する。
- (4) ビジネスロジックに設定された DAO 実装クラスを経由して、iBATIS の API を呼び出す。TERASOLUNA Server Framework for Java が提供する DAO 実装クラスのメソッドは各クラスの JavaDoc を参照のこと。
- (5) iBATIS は、ビジネスロジックから指定された SQLID をもとに iBATIS マッピング定義ファイルから SQL を取得する。
- (6) 取得した SQL を Bean 定義ファイルにて設定されたデータソースを使用してデータベースにアクセスする。

■ 使用方法

◆ コーディングポイント

- データソースの Bean 定義

データソースの設定はアプリケーション Bean 定義ファイルに定義する。

- アプリケーション Bean 定義ファイル

- ✧ JNDI の場合 JndiObjectFactoryBean を使用する。

Tomcat の場合、設定によってはデータソース名の頭に「java:comp/env/」を付加する必要があるため注意すること。

```
<bean id="TerasolunaDataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>java:comp/env/TerasolunaDataSource</value>
  </property>
</bean>
```

◇ JNDI 名が頻繁に変わる場合

context スキーマの<context:property-placeholder/>要素を使用して JNDI 名をプロパティファイルに記述する。

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:util="http://www.springframework.org/schema/util"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">
```

Property-placeholder を定義する。

<!-- JNDI 関連のプロパティ -->

```
<context:property-placeholder location="WEB-INF/jndi.properties"/>
```

```
<bean id="TerasolunaDataSource"
  class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>${jndi.name}</value>
  </property>
</bean>
```

context スキーマを定義する。

プロパティファイルの例 (jndi.properties)

```
jndi.name=java:comp/env/TerasolunaDataSource
```

- ◇ JNDI を使用しない場合は以下のように、DriverManagerDataSource を使用する。環境によって変換する DB 接続のための設定項目は、プロパティファイルに外出しにすることが望ましい。この場合、以下のように context スキーマの<context:property-placeholder/>要素を使用する。

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:util="http://www.springframework.org/schema/util"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">
```

Property-placeholder を定義する。

<!-- JDBC関連のプロパティ -->

● <context:property-placeholder location="WEB-INF/jdbc.properties" />

スキーマを定義する。

```
<bean id="TerasolunaDataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource"
  destroy-method="close">
  <property name="driverClassName" value="\${jdbc.driverClassName}" />
  <property name="url" value="\${jdbc.url}" />
  <property name="username" value="\${jdbc.username}" />
  <property name="password" value="\${jdbc.password}" />
</bean>
```

DriverManagerDataSource
の定義。

クラスパスを指定する。

※ 複数データソースを定義する場合は、bean 要素の id 属性を別の値で定義する。

➤ プロパティファイルの例 (jdbc.properties)

```
jdbc.driverClassName=oracle.jdbc.driver.OracleDriver
jdbc.url=jdbc:oracle:thin:@192.168.0.100:1521:ORCL
jdbc.username=name
jdbc.password=password
```


- iBATIS マッピング定義ファイル

このファイルは、ビジネスロジックで利用する SQL 文と、その SQL の実行結果を JavaBean にマッピングするための定義を設定する。また、モジュール単位にファイルを作成すること。ただし、SQL の ID は、アプリケーションで一意にする必要がある。SQL の詳細な記述方法に関しては、iBATIS のリファレンスを参照のこと。

- Select 文の実行例

- ✧ Select 文の実行には、<select>要素を使用する。
- ✧ resultClass 属性に SQL の結果を格納するクラスを指定する。結果が複数の場合は、指定したクラスの Collection あるいは配列が返却される。

```
<select id="getUser"
      resultClass="jp.terasoluna.....service.bean.UserBean">
    SELECT ID, NAME, AGE, BIRTH FROM USERLIST WHERE ID = #ID#
</select>
```

- Insert、Update、Delete 文の実行

- ✧ Insert 文の実行には、<insert>要素を使用する。
- ✧ Update 文の実行には、<update>要素を使用する。
- ✧ Delete 文の実行には、<delete>要素を使用する。
- ✧ parameterClass 属性に、登録するデータを保持しているクラスを指定する。parameterClass 属性に指定したクラス内のプロパティ名の前後に「#」を記述した部分に、値が埋め込まれた SQL 文が実行させる。

```
<insert id="insert_User"
      parameterClass="jp.terasoluna.....service.bean.UserBean">
    INSERT INTO USERLIST (ID, NAME, AGE, BIRTH ) VALUES (
      #id#, #name#, #age#, #birth#)
</insert>
```

- Procedure 文の実行

- ✧ Procedure 文の実行には、<procedure>要素を使用する。
- ✧ 基本的な使用方法は、他の要素と同様だが、<select>要素より、属性が少なくなっている。
- ✧ 値の設定、結果の取得は他の要素と異なり、parameterMap 属性、および<parameterMap>要素を使用する。

```
<sqlMap namespace="user">
  <parameterMap id="UserBean" class="java.util.HashMap">
    <parameter property="inputId" jdbcType="NUMBER"
      javaType="java.lang.String" mode="IN"/>
    <parameter property="name" jdbcType="VARCHAR"
      javaType="java.lang.String" mode="OUT"/>
  </parameterMap>
  <procedure id="selectUserName" parameterMap="user.UserBean">
    {call SELECTUSERNAME(?,?)}
  </procedure>
```

- iBATIS 設定ファイル

iBATIS 設定ファイルには、iBATIS の設定を記述することができるが、TERASOLUNA Server Framework for Java を使用する場合は、iBATIS 設定ファイルには、iBATIS マッピング定義ファイルの場所の指定のみ記述する。データソース、トランザクション管理は、Spring との連携機能を利用して行うため、本ファイルでその設定はしないこと。

- <sqlMap>要素は複数記述することができるため、iBATIS マッピング定義ファイルを分割した際は、複数記述すること。
- <settings>要素の useStatementNamespaces 属性は、SQLID を指定する際に完全修飾名で指定するかどうかを指定する。"true"を指定した場合は、『名前空間 + "." + SQLID』の形で SQLID を指定する。（例：名前空間が"user"、SQLID が"user.getUser"の場合、"user.getUser"と指定する）

- SqlMapClientFactoryBean の Bean 定義

Spring で iBATIS を使用する場合、SqlMapClientFactoryBean を使用して iBATIS 設定ファイルの Bean 定義を DAO に設定する必要がある。SqlMapClientFactoryBean は、iBATIS のデータアクセス時に利用されるメインのクラス「SqlMapClient」を管理する役割を持つ。

iBATIS 設定ファイルの Bean 定義はアプリケーション内で一つとする。

- iBATIS 設定ファイル

- ◇ “configLocation” プロパティに、iBATIS 設定ファイルのコンテキストルートからのパスを指定する。
- ◇ 単一データベースの場合は、“dataSource” プロパティに、使用するデータソースの Bean 定義を設定する。

```
<bean id="sqlMapClient"
      class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
  <property name="configLocation" value="WEB-INF/sqlMapConfig.xml"/>
  <property name="dataSource" ref="TerasolunaDataSource"/>
</bean>
```

- ◇ 複数データベースの場合は“dataSource” プロパティは指定せずに、“configLocation” プロパティのみ設定する。

```
<bean id="sqlMapClient"
      class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
  <property name="configLocation" value="WEB-INF/sqlMapConfig.xml"/>
</bean>
```

- DAO の Bean 定義

- DAO 実装クラスは、基本的にアプリケーション Bean 定義ファイルに定義する。また、DAO もトランザクション設定対象の Bean である。トランザクションの設定方法は『CA-01 トランザクション管理機能』の機能説明書を参照のこと。

また、Bean 定義時に DAO 実装クラスの “sqlMapClient” プロパティに iBATIS 設定ファイルの Bean 定義を設定する必要がある。

```
<bean id="sqlMapClient"
      class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
  <property name="configLocation" value="WEB-INF/sqlMapConfig.xml"/>
  <property name="dataSource" ref="TerasolunaDataSource"/>
</bean>
<bean id="queryDAO"
      class="jp.terasoluna.fw.dao.ibatis.QueryDAOiBatisImpl">
  <property name="sqlMapClient" ref="sqlMapClient"/>
</bean>
```

- 複数データベースの場合は、“sqlMapClient” プロパティの設定だけでなく、“dataSource” プロパティに、DAO 実装クラスで使用するデータソースを指定する必要がある。

```
<bean id="queryDAO"
      class="jp.terasoluna.fw.dao.ibatis.QueryDAOiBatisImpl">
  <property name="sqlMapClient" ref="sqlMapClient"/>
  <property name="dataSource" ref="TerasolunaDataSource"/>
</bean>
```

- QueryDAOiBatisImpl のメソッドの戻り値の指定

- 検索結果が1件または複数件の配列で、QueryDAOiBatisImpl の以下のメソッドを使用する場合は、戻り値の型と同じ型のクラスを引数に渡す必要がある。これにより、ビジネスロジックでのクラスキャストエラーの発生を避けることができる (DAO が `IllegalClassTypeException` をスローする)。

- ✧ `executeForObject (String sqlID, Object bindParams, Class clazz)`
- ✧ `executeForObjectArray (String sqlID, Object bindParams, Class clazz)`
- ✧ `executeForObjectArray (String sqlID, Object bindParams, Class clazz, int beginIndex, int maxCount)`

```
UserBean bean = dao.executeForObject("getUser", null, UserBean.class);  
UserBean[] bean  
    = dao.executeForObjectArray("getUser", null, UserBean[].class);
```

- 検索結果が複数件の Listで、QueryDAOiBatisImpl の以下のメソッドを使用する場合は、配列の場合と違って、戻り値の型のクラスを引数に渡さない。そのため、配列の場合に行っていた型の保証がされない点に注意する必要がある。

- ✧ `executeForObjectList (String sqlID, Object bindParams)`
- ✧ `executeForObjectList (String sqlID, Object bindParams, int beginIndex, int maxCount)`

```
List bean = dao.executeForObjectList("getUser", null);
```

- QueryDAOiBatisImpl を使用した一覧データ取得例

QueryDAOiBatisImpl を使用して、常に一覧情報（1 ページ分）をデータベースから取得する場合の設定およびコーディング例を以下に記述する。

- ① DAO 実装クラスを以下のように Bean 定義ファイルに定義する。

- Bean 定義ファイル

```
<bean id="queryDAO"  
      class="jp.terasoluna.fw.dao.ibatis.QueryDAOiBatisImpl">  
  <property name="sqlMapClient" ref="sqlMapClient"/>  
</bean>
```

- ② データアクセスを行うビジネスロジックの Bean 定義に、①で定義した DAO 実装クラスを設定する。なお、ビジネスロジックには DI コンテナより DAO を設定するための属性およびその Setter を用意しておく必要がある。

➤ Bean 定義ファイル

```
<bean id="listBLogic" scope="prototype"
      class="jp.terasoluna.sample.service.blogic.ListBLogic">
  <property name="queryDAO" ref="queryDAO" />
</bean>
```

➤ ビジネスロジック

Bean 定義ファイルにて設定された DAO の `executeForObjectArray(String sqlID, Object bindParams, Class clazz, int beginIndex, int maxCount)` メソッドを使用する。メソッドの引数に、アクションフォームに定義した「開始インデックス」と「表示行数」を設定する必要がある。

一覧表示の詳細な使用方法は、『WI-01 一覧表示機能』を参照のこと。

```
private QueryDAO queryDAO = null;
.....Setterは省略

public UserBean[] getUserList(ListBean bean) {
    int startIndex = bean.getStartIndex();
    int row = bean.getRow();
    UserBean[] bean = queryDAO.executeForObjectArray(
        "getUserList", null, UserBean.class, startIndex, row);
    return bean;
}
```

設定された DAO を使用して、データベースから一覧情報を取得する。

配列ではなく `java.util.List` の型で取得する場合、`executeForObjectList(String sqlID, Object bindParams, int beginIndex, int maxCount)` メソッドを使用する。

```
private QueryDAO queryDAO = null;
.....Setterは省略

public List<UserBean> getUserList(ListBean bean) {
    int startIndex = bean.getStartIndex();
    int row = bean.getRow();
    List<UserBean> bean = queryDAO.executeForObjectList(
        "getUserList", null, startIndex, row);
    return bean;
}
```

設定された DAO を使用して、データベースから一覧情報を取得する。

- UpdateDAOiBatisImpl を使用したデータ登録例

UpdateDAOiBatisImpl を使用して、データベースに情報を登録する場合の設定およびコーディング例を以下に記述する。Bean 定義ファイルの定義・設定方法は、QueryDAOiBatisImpl と同様なため省略する。

- ビジネスロジック

Bean 定義ファイルにて設定された DAO のメソッドを使用する。

```
private UpdateDAO updateDAO = null;

.....Setterは省略

public void register(UserBean bean) {
    .....
    updateDAO.execute("insertUser", bean);
    .....
}
```

設定された DAO を使用して、データベースにデータを登録する。

- UpdateDAOiBatisImpl を使用した複数データの登録例（オンラインバッチ処理）

UpdateDAOiBatisImpl を使用して、データベースに複数の情報を登録する場合の設定およびコーディング例を以下に記述する。Bean 定義ファイルの定義・設定方法は、QueryDAOiBatisImpl と同様なため省略する。

詳細は UpdateDAOiBatisImpl の JavaDoc を参照のこと。

- ビジネスロジック

Bean 定義ファイルにて設定された DAO の executeBatch(List<SqlHolder>)メソッドを使用する。

```
UserBean[] bean = map.get("userBeans");
List<SqlHolder> sqlHolders = new ArrayList<SqlHolder>();
for (int i = 0; i < bean.length; i++) {
    sqlHolders.add(new SqlHolder("insertUser", bean[i]));
}
updateDAO.executeBatch(sqlHolders);
.....
```

更新対象の sqlId、パラメータとなるオブジェクトを保持した SqlHolder のリストを作成する。

- 注意点

executeBatch は iBATIS のバッチ実行機能を使用している。executeBatch は戻り値として、SQL の実行によって変更された行数を返却するが、java.sql.PreparedStatement を使用しているため、ドライバにより正確な行数が取得できないケースがある。変更行数が正確に取得できないドライバを使用する場合、変更行数がトランザクションに影響を与える業務では（変更行数が 0 件の場合エラー処理をするケース等）、バッチ更新は使用しないこと。参考資料）

http://otndnld.oracle.co.jp/document/products/oracle11g/111/doc_dvd/java.111/E05720-02/oraperf.htm

「標準バッチ処理の Oracle 実装の更新件数」を参照のこと。

- StoredProcedureDAOiBatisImpl を使用したデータ取得例

StoredProcedureDAOiBatisImpl を使用して、データベースから情報を取得する場合の設定およびコーディング例を以下に記述する。Bean 定義ファイルの定義・設定方法は、QueryDAOiBatisImpl と同様なため省略する。

- ビジネスロジック

Bean 定義ファイルにて設定された DAO のメソッドを使用する。

```
private StoredProcedureDAO spDAO = null;

.....Setterは省略

public boolean register(UserBean bean) {
    .....
    Map<String, String> params = new HashMap<String, String>();
    params.put("inputId", bean.getId());
    spDAO.executeForObject("selectUserName", params);
    .....
}
```

設定された DAO を使用してプロシージャを実行する。

- iBATIS マッピング定義ファイル

- ✧ プロシージャの入出力を格納するための設定を<parameterMap>要素にて記述する。jdbcType 属性を指定すること。詳細な設定方法は、iBATIS のリファレンスを参照のこと。
参考資料)

http://ibatis.apache.org/docs/java/pdf/iBATIS-SqlMaps-2_ja.pdf

```
<sqlMap namespace="user">
  <parameterMap id="UserBean" class="java.util.HashMap">
    <parameter property="inputId" jdbcType="NUMBER"
      javaType="java.lang.String" mode="IN"/>
    <parameter property="name" jdbcType="VARCHAR"
      javaType="java.lang.String" mode="OUT"/>
  </parameterMap>
  <procedure id="selectUserName" parameterMap="user.UserBean">
    {call SELECTUSERNAME(?,?)}
  </procedure>
```

- 実行するプロシージャ (Oracle を利用した例)

```
CREATE OR REPLACE PROCEDURE SELECTUSERNAME
(inputId IN NUMBER, name out VARCHAR2) IS
BEGIN
  SELECT name INTO name FROM userList WHERE id = inputId ;
END ;
```


- QueryRowHandleDAOiBatisImpl を使用したデータ取得例
QueryRowHandleDAOiBatisImpl を使用して、データベースから情報を取得する場合の設定およびコーディング例を以下に記述する。Bean 定義ファイルの定義・設定方法は、QueryDAOiBatisImpl と同様なため省略する。

➤ DataRowHandler の実装

```
import jp.terasoluna.fw.dao.event.DataRowHandler;
```

```
public class SampleRowHandler implements DataRowHandler {
```

```
    public void handleRow(Object param) {
```

```
        if (param instanceof HogeData) {
```

```
            HogeData hogeData = (HogeData)param;
```

```
            // 一件のデータを処理するコードを記述
```

```
        }
```

```
    }
```

```
}
```

一件毎に handleRow メソッドが呼ばれ、引数に一件分のデータが格納されたオブジェクトが渡される。

一件のデータを元に更新処理を行うのであれば、あらかじめ DataRowHandler に UpdateDAO を渡しておく。
ダウンロードであれば ServletOutputStream などを渡しておくといよい。

➤ ビジネスロジック

```
private QueryRowHandleDAO queryRowHandleDAO = null;
```

```
.....Setterは省略
```

```
public BLogicResult execute(BLogicParam params) {
```

```
    Parameter param = new Parameter();
```

```
    HogeDataRowHandler dataRowHandler = new HogeDataRowHandler();
```

```
    queryRowHandleDAO.executeWithRowHandler(
```

```
        "selectDataSql", param, dataRowHandler);
```

```
    BLogicResult result = new BLogicResult();
```

```
    result.setResultString("success");
```

```
    return result;
```

```
}
```

実際に一件ずつ処理を行う DataRowHandler インスタンスを渡す。

◆ 拡張ポイント

なし。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.dao.QueryDAO	参照系 SQL を実行するための DAO インタフェース
2	jp.terasoluna.fw.dao.UpdateDAO	更新系 SQL を実行するための DAO インタフェース
3	jp.terasoluna.fw.dao.StoredProcedureDAO	StoredProcedure を実行するための DAO インタフェース
4	jp.terasoluna.fw.dao.QueryRowHandleDAO	参照系 SQL を実行し一件ずつ処理するための DAO インタフェース
5	jp.terasoluna.fw.dao.ibatis.QueryDAOiBatisImpl	QueryDAO インタフェースの iBATIS 用実装クラス
6	jp.terasoluna.fw.dao.ibatis.UpdateDAOiBatisImpl	UpdateDAO インタフェースの iBATIS 用実装クラス
7	jp.terasoluna.fw.dao.ibatis.StoredProcedureDAOiBatisImpl	StoredProcedureDAO インタフェースの iBATIS 用実装クラス
8	jp.terasoluna.fw.dao.ibatis.QueryRowHandleDAOiBatisImpl	QueryRowHandleDAO インタフェースの iBATIS 用実装クラス

■ 関連機能

- 『CA-01 トランザクション管理機能』
- 『WI-01 一覧表示機能』

■ 使用例

- Terasoluna Server Framework for Java (Web 版) チュートリアル
 - 「2.6 データベースアクセス」
 - 「2.7 登録」
 - 一覧表示画面、登録画面
- Terasoluna Server Framework for Java (Rich 版) チュートリアル
 - 「2.4 データベースアクセス」
 - /webapps/WEB-INF/dataAccessContext-local.xml
 - /webapps/WEB-INF/sql-map-config.xml
 - /sources/sqlMap.xml
 - jp.terasoluna.rich.tutorial.service.blogic.DBAccessBLogic.java 等

- Terasoluna Server Framework for Java (Web 版) 機能網羅サンプル

- 「UC01 データベースアクセス」

- ◇ /webapps/database/*
- ◇ /webapps/WEB-INF/database/*
- ◇ jp.terasoluna.thin.functionsample.database.*

■ 備考

- 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。

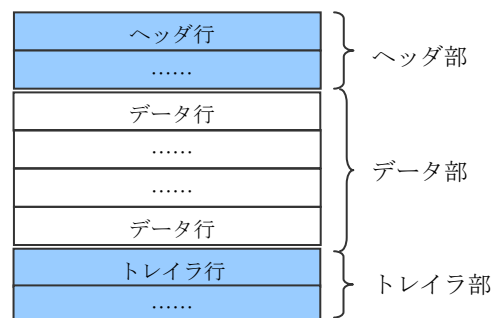
※例えば PostgreSQL JDBC ドライバ(postgresql-8.3-604.jdbc3.jar にて確認) のデフォルトは、一度の通信で検索対象のデータが全件取得される。数十件程度の件数であれば問題にならないが、数万件以上の大量データを検索する場合にはヒープメモリを圧迫してしまう可能性がある。

BC-01 ファイルアクセス機能

■ 概要

◆ 機能概要

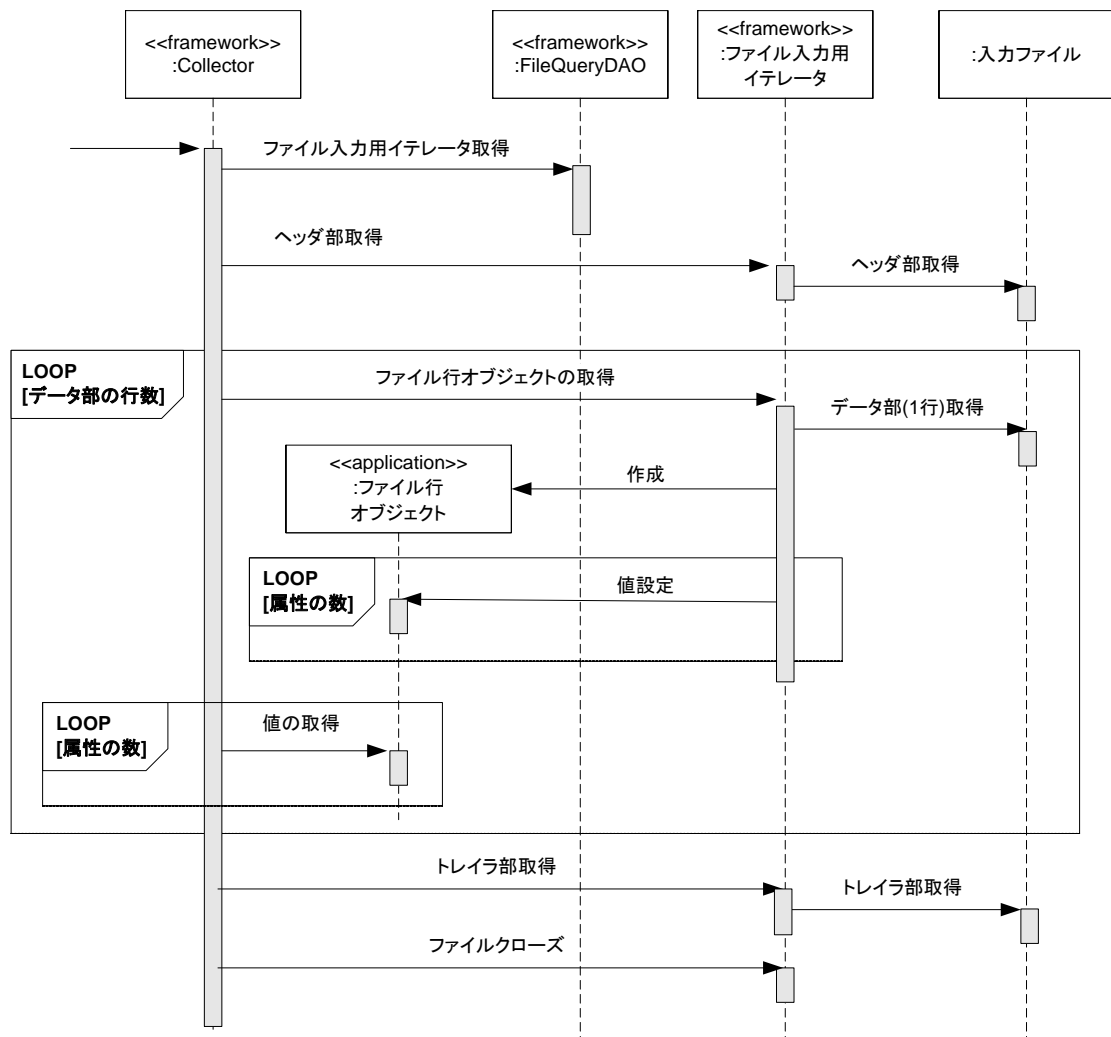
- CSV 形式、固定長形式、可変長形式ファイルの入出力機能を提供する。
 - ファイル入力機能は順次読込のみ提供する。
- ファイルアクセス機能で対象とするファイル構造は下図のとおりである。
 - ヘッダ部、トレイラ部の無いファイルについては、ヘッダ部、トレイラ部を 0 行とする。
 - データ部のデータ構造はファイル行オブジェクト（POJO）にアノテーションを使用して定義する。



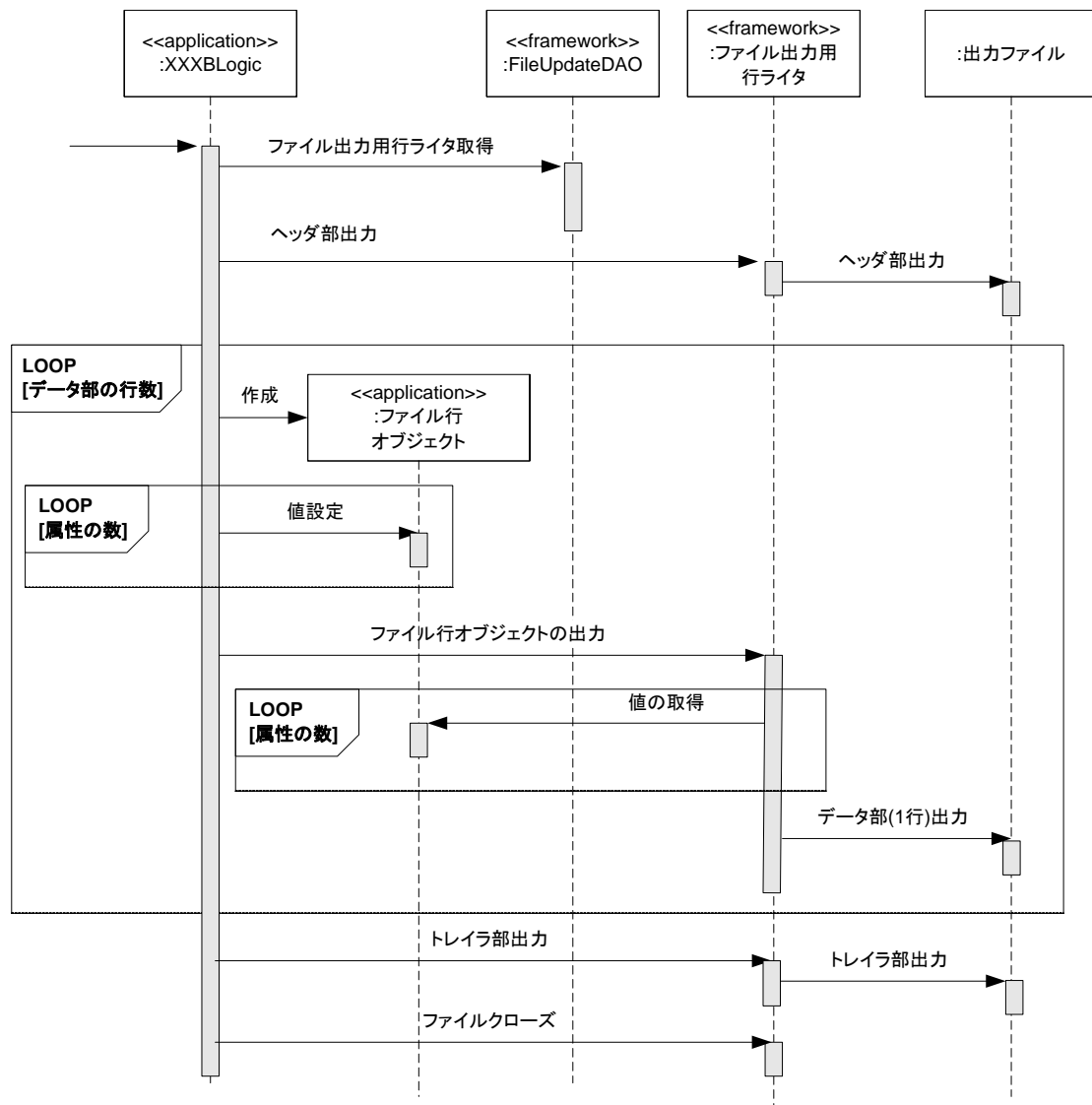
- 入出力データに対するフォーマット処理を行う。
 - ファイル行オブジェクト（POJO）の定義により、項目に対するパディング（Padding）、トリム（Trim）、文字変換（StringConverter）等が指定できる。

◆ 概念図

- ファイル入力処理の概念図（Collector から利用される場合）



- ファイル出力処理の概念図（ビジネスロジックから利用される場合）



◆ 解説

- ファイルアクセス機能で取り扱うファイル
TERASOLUNA-Batch のファイルアクセス機能は CSV 形式、固定長形式、可変長形式、文字列データのファイル入出力機能を提供する。ファイル内の各行の項目数及び項目の並び順は同一である必要がある。
 - CSV 形式
CSV 形式とは、データを「,(カンマ)」で区切ったものである。データを区切る際に使用しているカンマを特に"区切り文字"と呼ぶ。CSV 形式は可変長ファイルの区切り文字を「,(カンマ)」に固定したものになる。
 - 固定長形式
固定長形式とは、対象データの 1 行の各項目の長さ（バイト数）が全ての行で同じであるもの。対象データを区切る方法は、各項目に割り当てられているバイト数をもとに行う。
 - 可変長形式
可変長形式とは、対象データの 1 行の各項目の長さ（バイト数）が可変である（異なる）もの。対象データは"区切り文字" を使って区切る。
 - 文字列データ
文字列データとは、区切り文字やバイト数でデータを区切る必要が無いもの。1 行分のデータを String 型として扱う。
- ファイル行オブジェクト
ファイル入出力を使用する場合には、ファイルのデータ部の 1 行分のデータに対応するように Java Bean のクラスを作成する。作成する Java Bean のクラスを、本フレームワークではファイル行オブジェクトと呼ぶ。
ファイル行オブジェクトのクラスには、ファイル全体に関わる定義情報（改行文字等）を Java アノテーションにより設定する。
ファイル行オブジェクトのクラスが持つ属性には、ファイルの個々の項目の定義情報（バイト長等）を Java アノテーションにより設定する。

- ファイル全体に関わる定義情報（FileFormat アノテーション）
 ファイル全体に関わる定義情報は、ファイル行オブジェクトのクラスに対してアノテーション FileFormat により設定する。FileFormat アノテーションは入力ファイル、および出力ファイルのどちらの場合にも同じアノテーションを設定する。
 ➤ アノテーション FileFormat 設定項目

項番	論理項目名	説明	デフォルト値	CSV		固定長		可変長		文字列データ	
	物理項目名			入	出	入	出	入	出	入	出
1	行区切り文字	行区切り文字(改行文字)を設定する。	システムデフォルト/なし(固定長)	○	○	○	○	○	○	○	○
	lineFeedChar										
2	区切り文字	「,(カンマ)」等の区切り文字を設定する。	「,(カンマ)」	×	×	×	×	○	○		
	delimiter										
3	囲み文字	「“(ダブルクォーテーション)”等のカラムの囲み文字を設定する。	なし	○	○	×	×	○	○		
	encloseChar										
4	ファイルエンコーディング	入出力を行うファイルのエンコーディングを設定する。	システムデフォルト	○	○	○	○	○	○	○	○
	fileEncoding										
5	ヘッダ行数	入力ファイルのヘッダ部に相当する行数を設定する。	0	○		○		○		○	
	headerLineCount										
6	トレイラ行数	入力ファイルのトレイラ部に相当する行数を設定する。	0	○		○		○		○	
	trailerLineCount										
7	ファイル上書きフラグ	出力ファイルと同じ名前のファイルが存在する場合に上書きするかどうかを設定する。 [true/false]（上書きする/上書きしない）	FALSE		○		○		○		○
	overWriteFlg										

※○の項目は必要に応じて設定可。×の項目は設定できないことを表している（×の項目を設定した場合、実行時にエラーとなる）。無印は設定を無視することを表している。

※「行区切り文字」の"システムデフォルト"とは、
 System.getProperty("line.separator");で取得できる実行環境に依存した値である。
 固定長形式の場合のみ、デフォルト値はなし。

※固定長形式で行区切り文字を無し（改行無し）とした場合、ヘッダ行数とトレイラ行数を設定することはできない。

※「ファイルエンコーディング」の"システムデフォルト"とは、
 System.getProperty("file.encoding");で取得できる実行環境に依存した値である。

- ※「行区切り文字」、「区切り文字」でタブ、改行文字を使用する場合、Java 言語仕様で定められているエスケープシーケンス（`\t`、`\r` 等）で記述すること。
- ※可変長の「区切り文字」として「`\u0000`」を設定することはできない。
- ※「区切り文字」と「囲み文字」は同一の値を設定することができない。
- ※「行区切り文字」と「区切り文字」は同一の値を設定することができない。

- ファイル項目の定義情報

(InputFileColumn アノテーション、OutputFileColumn アノテーション)

ファイル項目の定義情報は、ファイル行オブジェクトのクラスが持つ属性に対してアノテーション InputFileColumn、OutputFileColumn により設定する。アノテーション InputFileColumn、OutputFileColumn は入力ファイル、および出力ファイルのどちらの場合にも同じアノテーションを設定する。

ひとつのファイル行オブジェクトが入力ファイル、および出力ファイルの両方で使用される場合には、ひとつの属性に対して二つのアノテーション

(InputFileColumn、OutputFileColumn) を設定する。

➤ InputFileColumn, OutputFileColumn の設定項目

項番	論理項目名	説明	デフォルト値	CSV		固定長		可変長	
	物理項目名			入	出	入	出	入	出
1	カラムインデックス	データ部の 1 行のカラムの内、何番目のデータをファイル行オブジェクトの属性に格納するのかを設定する。インデックスは「0(ゼロ)」から始まる整数。	なし	◎	◎	◎	◎	◎	◎
	columnIndex								
2	フォーマット	BigDecimal 型、Date 型に対するフォーマットを設定する。	なし	○	○	○	○	○	○
	columnFormat								
3	バイト長	各カラムに対するバイト長を設定する。	なし	○	○	◎	◎	○	○
	bytes								
4	パディング種別	パディングの種別を設定する。列挙型 PaddingType から値を選択する。[RIGHT/LEFT/NONE] (右寄せ/左寄せ/パディングなし)	NONE		○		○		○
	paddingType								
5	パディング文字	パディングする文字を設定する(半角文字のみ設定可能)。	なし		○		○		○
	paddingChar								
6	トリム種別	トリムの種別を設定する。列挙型 TrimType から値を選択する。[RIGHT/LEFT/BOTH /NONE] (右寄せ/左寄せ/両側/トリムなし)	NONE	○	○	○	○	○	○
	trimType								
7	トリム文字	トリムする文字を設定する(半角文字のみ設定可能)。	なし	○	○	○	○	○	○
	trimChar								
8	文字変換種別	String 型のカラムについて、大文字変換等を設定する。StringConverter インタフェースの実装クラスを指定する。 StringConverterToUpperCase.class (大文字に変換) / StringConverterToLowerCase.class (小文字に変換) / NullStringConverter.class (変換しない)	NullStringConverter.class	○	○	○	○	○	○
	stringConverter								
9	囲み文字	カラム単位で「“(ダブルクォーテーション)”等のカラムの囲み文字を設定する。	なし	○	○	×	×	○	○
	columnEncloseChar								

※ ◎の項目はアノテーションを設定する際の必須項目(必須項目を設定しなかった場合、実行時にエラーとなる)。○の項目は必要に応じて設定可。無印は設定を行っても有効にならないことを表している。

- ※ バイト長とは、入力時はファイルから取得する時点の長さであり、各種変換処理後の長さとは異なる。出力時はファイルへ出力する時点の長さであり、各種変換処理後の長さである。
- ※ パディング種別、トリム種別を指定したときには、それぞれパディング文字、トリム文字を必ず設定すること。
- ※ パディング種別で **NONE** 以外を指定したときはバイト長を必ず設定すること。ここでのバイト長は、パディング処理を行った後のバイト長を設定すること。
- ※ 入力処理時にパディングを設定した場合、取得データがバイト長で設定した長さ以外の場合はバイト長チェックでエラーが発生し、バイト長で設定した長さと一致する場合はパディングすべきデータ数が 0 となるため、パディング処理を行っても取得データと同じになる。つまり、入力時にパディングの設定を行っても有効にならないことに留意すること。
- ※ 変換処理の順番は、入力時と出力時で異なることに留意すること。
 - ・ 入力時はバイト数チェック、トリム処理、パディング処理、文字変換処理である。
 - ・ 出力時はトリム処理、パディング処理、文字変換処理、バイト数チェックである。
- ※ **FileFormat** の **encloseChar** と **InputFileColumn** または **OutputFileColumn** の **columnEncloseChar** の両方が設定されている場合、**columnEncloseChar** の設定が優先される
- ※ **CSV** 形式、可変長形式の場合、ファイル行オブジェクトに **InputFileColumn**、**OutputFileColumn** が一つも設定されていない場合、実行時にエラーとなる。

- データ項目定義と異なるデータを入出力した場合の例外処理
対象データの入力の際、アノテーションの記述と異なるデータがあった場合、フレームワークは例外を発生させる。例外が発生する例としては、日付型のフォーマットを設定しているところに、数値型のデータを格納しようとした場合などが挙げられる。
- ファイル行オブジェクトの属性について
 - ✧ ファイル行オブジェクトで使える属性の型は、`java.lang.String`、`int`、`java.math.BigDecimal`、`java.util.Date` の 4 種類とする。
 - ✧ 各属性には値を操作するために可視性が `public` である `setter/getter` を用意すること。
- フォーマットについての補足
数値型、日付型の入出力は、フォーマットとして入力した文字列に沿ってデータの入出力を行う。詳細については、**Java Platform Standard Edition (Java SE) API 仕様** を参照のこと。
"java.text.DecimalFormat"、"java.text.SimpleDateFormat"
- ファイルの 1 行あたりのカラム数についての補足
ファイル入力の際、アノテーションを設定したカラム数とファイルのカラム数が異なる場合、フレームワークは例外を発生させる。また、固定長ファイル入力の際、アノテーションの `bytes` で設定したカラムのバイト数の合計と読み取った 1 行のバイト数が異なる場合、フレームワークは例外を発生させる。
- 囲み文字についての補足
囲み文字が出力項目に含まれている場合、同じ囲み文字を追加してエスケープ編集を行う。また、ファイル入力の際、カラムに囲み文字と同一の文字が含まれておりエスケープ編集されていないカラムの場合、フレームワークは例外を発生させる。
- ファイル上書きフラグについての補足
`FileUpdateDAO` を DI したビジネスロジック分割ジョブで起動した場合、ファイル上書きフラグを `True` に設定するとデータが破損する可能性があることに留意すること。また、分割ジョブにてファイル上書きファイルフラグを `false` にした場合、複数スレッドから一つのファイルにアクセスするため、データの出力順番がランダムになることに留意すること。

- ファイル入力用 DAO

フレームワークではファイル入力用 DAO インタフェース、およびファイル入力用イテレータインタフェースを規定し、ファイル形式に対応したそれぞれのデフォルト実装を提供する。

ファイル入力用 DAO の `execute()` メソッドを実行し、ファイル入力用イテレータを取得する。ファイルの各行は、ファイル入力用イテレータの `next()` メソッドで取得する。

- ファイル入力用インタフェース

項番	インタフェース名	概要
1	jp.terasoluna.fw.file.dao.FileQueryDAO	ファイル入力用 DAO インタフェース
2	jp.terasoluna.fw.file.dao.FileLineIterator	ファイル入力用イテレータインタフェース

- ◇ ファイル入力用 DAO 実装クラス

項番	クラス名	概要
1	jp.terasoluna.fw.file.dao.standard.CSVFileQueryDAO	CSV 形式のファイル入力を行う場合に利用する
2	jp.terasoluna.fw.file.dao.standard.FixedFileQueryDAO	固定長形式のファイル入力を行う場合に利用する
3	jp.terasoluna.fw.file.dao.standard.VariableFileQueryDAO	可変長形式のファイル入力を行う場合に利用する
4	jp.terasoluna.fw.file.dao.standard.PlainFileQueryDAO	文字列データをファイルから入力する場合に利用する

- ・ ファイル入力用 DAO は、ファイル入力用イテレータを生成する。

- ◇ ファイル入力用イテレータ実装クラス

項番	クラス名	概要
1	jp.terasoluna.fw.file.dao.standard.CSVFileLineIterator	CSV 形式のファイル入力を行う場合に利用する
2	jp.terasoluna.fw.file.dao.standard.FixedFileLineIterator	固定長形式のファイル入力を行う場合に利用する
3	jp.terasoluna.fw.file.dao.standard.VariableFileLineIterator	可変長形式のファイル入力を行う場合に利用する
4	jp.terasoluna.fw.file.dao.standard.PlainFileLineIterator	文字列データをファイルから入力する場合に利用する

- ・ ファイル入力でのデータ部の入力は、データ部の 1 行分のデータを入出力オブジェクトに格納し、呼び出し元に返却する処理を提供する。
- ・ ヘッダ部、トレイラ部からの入力用メソッドを提供する。
- ・ 囲み・区切り文字として設定された文字が入力データの文字列にあると正しく動作しない。

- ファイル出力用 DAO

フレームワークではファイル出力用 DAO、およびファイル出力用行ライタのインタフェースを規定し、ファイル形式に対応したそれぞれのデフォルト実装を提供する。

ファイル出力用 DAO の `execute()` メソッドを実行し、ファイル出力用行ライタを取得する。ファイルの各行は、ファイル出力用行ライタの `printDataLine()` メソッドで出力する。

- ファイル出力用インタフェース

項番	インタフェース名	概要
1	jp.terasoluna.fw.file.dao.FileUpdateDAO	ファイル出力用 DAO インタフェース
2	jp.terasoluna.fw.file.dao.FileLineWriter	ファイル出力用行ライタインタフェース

- ◇ ファイル出力用 DAO 実装クラス

項番	クラス名	概要
1	jp.terasoluna.fw.file.dao.standard.CSVFileUpdateDAO	CSV 形式のファイル出力を行う場合に利用する
2	jp.terasoluna.fw.file.dao.standard.FixedFileUpdateDAO	固定長形式のファイル出力を行う場合に利用する
3	jp.terasoluna.fw.file.dao.standard.VariableFileUpdateDAO	可変長形式のファイル出力を行う場合に利用する
4	jp.terasoluna.fw.file.dao.standard.PlainFileUpdateDAO	文字列データをファイルへ出力する場合に利用する

- ・ ファイル出力用 DAO は、ファイル出力用イテレータを生成する。

- ◇ ファイル出力用行ライタ実装クラス

項番	クラス名	概要
1	jp.terasoluna.fw.file.dao.standard.CSVFileLineWriter	CSV 形式のファイル出力を行う場合に利用する
2	jp.terasoluna.fw.file.dao.standard.FixedFileLineWriter	固定長形式のファイル出力を行う場合に利用する
3	jp.terasoluna.fw.file.dao.standard.VariableFileLineWriter	可変長形式のファイル出力を行う場合に利用する
4	jp.terasoluna.fw.file.dao.standard.PlainFileLineWriter	文字列データをファイルへ出力する場合に利用する

- ・ ファイル出力でのデータ部の出力は、ファイル行オブジェクトに格納された一行分のデータをファイルに書込む処理を提供する。
- ・ ヘッダ部、トレイラ部への出力メソッドを提供する
- ・ ファイル生成時、フォルダ名は存在するフォルダを設定する必要がある。存在しないとファイルは生成されない。

- ファイル入力チェックについて
Collector での対象データ取得時に、入力ファイルに対して入力チェックを行うことができる。
入力チェックについての詳細は、『BD-02 対象データ取得機能』を参照のこと。
- 例外処理
ファイルアクセス時に例外が発生した場合、ファイルアクセス用の DAO からスローする例外クラスに、エラーが発生したファイルの情報を格納する。例外が発生した処理に対する後処理(処理過程で生成されたファイルの削除処理など)は例外ハンドラで実装すること。例外ハンドラの詳細については『BH-01 例外ハンドリング機能』を参照のこと。
ファイルアクセス時の例外クラスには、以下の2つのクラスがある。

項番	例外クラス名	概要
1	jp.terasoluna.fw.file.dao. FileNotFoundException	ファイル全体に関わるエラーに対応する例外クラス。 以下の情報を保持する。 ・ファイル名
2	jp.terasoluna.fw.file.dao. FileLineException (FileNotFoundException のサブクラス)	ファイルの行に関わるエラーに対応する例外クラス。 以下の情報を保持する。 ・ファイル名 ・エラーが発生した箇所の行番号 ・エラーが発生したカラムのカラムインデックス (0 から開始) ・エラーが発生したカラムのカラム名(ファイル 行オブジェクトのプロパティ名)

例外クラスが保持する情報を用いて、ログ出力などを行うことができる。

■ 使用方法

◆ コーディングポイント

- ファイル行オブジェクトの実装例
 - CSV 形式のデータをファイル行オブジェクトに格納する場合の記述例 (getter/setter は省略)

```

@FileFormat(encloseChar = '"')
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;
    .....
}

```

アノテーションの FileFormat は必須

アノテーション InputFileColumn とパラメータの設定

- ◇ 上記のファイル行オブジェクトに下記の CSV 形式のデータを格納すると、各属性の値は以下の通りとなる。

"2006/07/01","shop01","1,000,000" ← CSV形式のデータ

- ◇ ファイル行オブジェクトに設定される値

```

hiduke = Sat Jul 01 00:00:00 JST 2006
shopId = SHOP01
uriage = 1000000

```

文字列データ (PlainFileQueryDAO, PlainFileUpdateDAO) を利用する場合は、@FileFormat のみを記述したファイル行オブジェクトを使用すること。

➤ ファイル全体に関わる定義情報を設定する場合の記述例 (getter/setter は省略)

```
@FileFormat(encloseChar = '"', lineFeedChar="\r\n",  
headerLineCount = 1, trailerLineCount= 1)
```

```
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;
```

```
.....
```

```
}
```

FileFormat で、ヘッダ部行数とトレイラ部行数を指定する。

◇ 上記のファイル行オブジェクトに下記のデータを格納すると、各属性の値は以下の通りとなる。

支店名 : 千葉支店

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

合計金額 : 1,000,000

← ヘッダ部

← データ部

← トレイラ部

ヘッダ部とトレイラ部を含んだファイル

◇ ファイル行オブジェクトに設定される値

hiduke = Sat Jul 01 00:00:00 JST 2006

shopId = SHOP01

uriage = 1000000

※ ヘッダ部とトレイラ部はファイル行オブジェクトに格納されない。

- ファイル項目でトリム種別を設定し、デフォルトのトリム文字を使用した定義情報を設定する場合の記述例 (getter/setter は省略)

```
@FileFormat(encloseChar = '"')
public class SampleFileLineObject {
    .....
    @InputFileColumn(
        columnIndex = 0,
        columnFormat="yyyy/MM/dd")
    private Date hiduke = null;

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

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

右側の空白をトリム(削除)するように設定する。

- ◇ 上記のファイル行オブジェクトに下記の CSV 形式のデータを格納すると、各属性の値は以下の通りとなる。

```
"2006/07/01","shop01 ", "1,000,000" ← データ部
```

- ◇ ファイル行オブジェクトに設定される値

```
hiduke = Sat Jul 01 00:00:00 JST 2006
shopId = SHOP01 ←右側にあった空白文字を削除している。
uriage = 1000000
```

- ファイル項目でトリム種別を設定し、個別のトリム文字を使用した定義情報を設定する場合の記述例 (getter/setter は省略)

```
@FileFormat(encloseChar = '')
public class SampleFileLineObject {
    .....
    @InputFileColumn(
        columnIndex = 0,
        columnFormat="yyyy/MM/dd")
    private Date hiduke = null;

    @InputFileColumn(
        columnIndex = 1,
        trimType = TrimType.LEFT,
        trimChar = '0',
        stringConverter = StringConverterToUpperCase.class)
    private String shopId = null;

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

左側の'0'の文字ををトリム(削除)するように設定する。

- ◇ 上記のファイル行オブジェクトに下記の CSV 形式のデータを格納すると、各属性の値は以下の通りとなる。

```
"2006/07/01","000shop01","1,000,000" ← データ部
```

- ◇ ファイル行オブジェクトに設定される値

```
hiduke = Sat Jul 01 00:00:00 JST 2006
shopId = SHOP01 ←対象文字列の左側にある'0'が削除される。
uriage = 1000000
```

● ジョブ Bean 定義ファイルの設定例

```
<bean id="CSVFile01"
    class="jp.terasoluna.XXX....">
    <!-- 入力ファイルの設定 -->
    <property name="fileDao">
        <ref bean="csvFileQueryDAO" />
    </property>
```

ファイル入出力用 DAO を使うクラス

ファイル入出力用 DAO 実装クラス。
参照する Bean は「FileAccessBean.xml」を参照のこと

- ファイル全体に囲み文字を設定し、更にファイル項目で個別の囲み文字を使用した定義情報を設定する場合の記述例 (getter/setter は省略)

```
@FileFormat(encloseChar = '"')
public class SampleFileLineObject {
    .....

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

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

    @InputFileColumn(
        columnIndex = 2,
        columnEncloseChar = '|',
        columnFormat="###,###,###")
    private BigDecimal uriage = null;
    .....
}
```

ファイル全体に囲み文字を設定する。

カラム単位で囲み文字を設定する。
(全体の設定より優先される)

- ◇ 上記のファイル行オブジェクトに下記の CSV 形式のデータを格納すると、各属性の値は以下の通りとなる。

```
"2006/07/01","shop01",|1,000,000| ← データ部
```

- ◇ ファイル行オブジェクトに設定される値

```
hiduke = Sat Jul 01 00:00:00 JST 2006
shopId = SHOP01
uriage = 1000000
```

- 固定長形式で行区切り無しのデータをファイル行オブジェクトに格納する場合の記述例(getter/setter は省略)

```
@FileFormat(lineFeedChar = "")
public class SampleFileLineObject {
    .....
    @InputFileColumn(
        columnIndex = 0,
        bytes = 10,
        columnFormat="yyyy/MM/dd")
    private Date hiduke = null;

    @InputFileColumn(
        columnIndex = 1,
        bytes = 6,
        stringConverter = StringConverterToUpperCase.class)
    private String shopId = null;

    @InputFileColumn(
        columnIndex = 2,
        bytes = 9,
        columnFormat="###,###,###")
    private BigDecimal uriage = null;
    .....
}
```

行区切り文字を無しに設定する。

各カラムにバイト数を設定する。

- ◇ 上記のファイル行オブジェクトに下記の固定長形式のデータを格納すると、各属性の値は以下の通りとなる。

2006/07/01shop011,000,000 ← データ部

- ◇ ファイル行オブジェクトに設定される値

```
hiduke = Sat Jul 01 00:00:00 JST 2006
shopId = SHOP01
uriage = 1000000
```

- ファイル入力の実装例

- ファイル入力処理の実装

- (1) ファイル行オブジェクトを実装する。
- (2) ファイル入力処理を行うクラスのプロパティに、FileQueryDAO 実装クラスを設定する。

【ジョブ Bean 定義ファイルの設定例】

```
<bean id="blogic"
      class="jp.terasoluna.batch.sample.SampleLogic">
  <property name="fileQueryDAO" ref="csvFileQueryDao" />
</bean>
```

参照する Bean は「FileAccessBean.xml」を参照のこと

- (3) ファイル入力処理を行うクラスでは、FileQueryDAO の execute() メソッドでファイル入力用イテレータを取得する。ファイル入力用イテレータ取得時に、ファイルオープンが行われる。
ファイル入力用イテレータの next メソッドで、ファイル行オブジェクトを取得する。

【実装例】

```
...
// ファイル入力用イテレータの取得
FileLineIterator<SampleFileLineObject> fileLineIterator
    = fileQueryDAO.execute(basePath +
        "/some_file_path/uriage.csv", FileColumnSa
        mple.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 FileFormat 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 FileFormat の trailerLineCount で設定した行数分のトレイラ部を取得する

closeFile()メソッドでファイルを閉じること

- ファイルの入力順序

トレイラ部の入力、データ部の入力が全て終わった後に行う必要がある点に留意すること。

➤ スキップ処理

ファイル入力機能では入力を開始する行を指定できる。これは主に『BE-04 リスタート機能』で中断したジョブを再開する際、リスタートポイントからファイルの読み込みを再開するために利用する。

【ビジネスロジックの実装例】

```
// スキップ処理
```

```
.....
```

```
    fileLineIterator.skip(1000);
```

```
.....
```

fileLineIterator のカレント行から 1000 行分の
データ行を読み飛ばす処理を行う

➤ ファイル入力処理の実装（ファイル入力を行うクラスでファイルをオープンしたまま、読み込みを行う場合）

ファイル入力処理を行うクラスに、(File 更新用 Dao ではなく) ファイル入力用行イテレータを直接設定する。ファイル入力用行イテレータのコンストラクタで、ファイルのパス、ファイル行オブジェクトのクラス等を指定する。

【ジョブ Bean 定義ファイルの設定例】

```
<bean id="blogic" class="testBlogic">
  <property name="iterator">
    <bean class="jp.terasoluna.fw.file.dao.standard.
      CSVFileLineIterator"
      destroy-method="closeFile">
      <constructor-arg index="0" value="some_file_path/uriage.csv" />
      <constructor-arg index="1"
        value="jp.terasoluna.batch.sample.FileColumnSample" />
      <constructor-arg index="2" ref="columnParserMap" />
    </bean>
  </property>
</bean>
```

bean のコンストラクタにファイル名(1 番目の引数。文字型)、パラメータクラス(2 番目の引数、クラス型)、テキスト変換処理(3 番目の引数。FileAccessBean.xml で定義されている Bean "columnParserMap"を固定で指定)を設定する。

➤ ファイル入力処理の実装についての補足

ファイル入力処理を行う場合は、FileQueryDAO を利用してもファイル入力用行イテレータを直接設定する利用する方法のどちらかを利用することで実装出来る。FileQueryDAO を利用する場合は DAO が呼ばれるたびにファイルのオープン/クローズ処理が行われる。そのため、ファイル入力用行イテレータを直接設定した場合と比べて処理が遅くなる。また、ファイル入力用行イテレータを直接設定した場合に、ビジネスロジックで入力処理を行い後処理でファイルの移動などを行うと後処理ではまだファイルストリームが存在するため、エラーが発生することに留意すること。

- ファイル出力の実装例

- ファイルの設定として、囲み文字と区切り文字を設定し、データの一部をデフォルトのパディング文字でパディング処理したデータをファイルに出力する場合の記述例 (getter/setter は省略)

```
@FileFormat(delimiter = ",", encloseChar = '"')
public class SampleFileLineObject {
    .....
    @OutputFileColumn(
        columnIndex = 0,
        columnFormat="yyyy/MM/dd")
    private Date hiduke = null;

    @OutputFileColumn(
        columnIndex = 1,
        paddingType = PaddingType.LEFT,
        bytes = 10,
        stringConverter = StringConverterToLowerCase.class)
    private String shopId = null;

    @OutputFileColumn(
        columnIndex = 2,
        columnFormat="###,###,###")
    private BigDecimal uriage = null;
    .....
}
```

アノテーションの FileFormat は必須
区切り文字、囲み文字を設定。

アノテーション OutputFileColumn
とパラメータの設定

パディング処理を行う場合は、バ
イト数の設定が必須となる。

◇ 出力対象となるファイル行オブジェクトの値

```
hiduke = Sat Jul 01 00:00:00 JST 2006
shopId = SHOP01
uriage = 1000000
```

◇ 上記のファイル行オブジェクトを出力すると以下の値となる。

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


- データの一部を個別のパディング文字でパディング処理したデータをファイルに出力する場合の記述例 (getter/setter は省略)

```
@FileFormat(delimiter = ",", encloseChar = '"')
public class SampleFileLineObject {
    .....
    @OutputFileColumn(
        columnIndex = 0,
        columnFormat="yyyy/MM/dd")
    private Date hiduke = null;

    @OutputFileColumn(
        columnIndex = 1,
        paddingType = PaddingType.RIGHT,
        paddingChar = '0',
        bytes = 10,
        stringConverter = StringConverterToLowerCase.class)
    private String shopId = null;

    @OutputFileColumn(
        columnIndex = 2,
        columnFormat="###,###,###")
    private BigDecimal uriage = null;
    .....
}
```

右側のパディング処理を行い、パディング文字として'0'を設定する。

◇ 出力対象となるファイル行オブジェクトの値

```
hiduke = Sat Jul 01 00:00:00 JST 2006
shopId = SHOP01
uriage = 1000000
```

◇ 上記のファイル行オブジェクトを出力すると以下の値となる。

```
"2006/07/01","shop010000","1,000,000"
```

➤ ファイル出力処理の実装（1 メソッドでファイルをオープン・クローズする場合）

- (1) ファイル行オブジェクトを実装する。
- (2) ファイル出力処理を行うクラスのプロパティに、FileUpdateDAO 実装クラスを設定する。

【ジョブ Bean 定義ファイルの設定例】

```
<bean id="blogic"
      bean class="jp.terasoluna.batch.sample.SampleLogic">
  <property name="fileUpdateDAO" ref="csvFileUpdateDao" />
</bean>
```

- (3) ファイル出力処理を行うクラスでは、FileUpdateDAO の execute メソッドでファイル出力用行ライタを取得する。ファイル出力用行ライタの取得時に、ファイルがオープンされる。

【実装例】

```
...
// ファイル出力用行ライタの取得
FileLineWriter< SampleFileLineObject > fileLineWriter
= fileUpdateDAO.execute(basePath + "/some_file_path/uriage.csv",
                        SampleFileLineObject.class);
...

try {
  // ヘッダ部の出力
  fileLineWriter.printHeaderLine(headerString);
  ...

  while ( ... ) {
    ...
    // データ部の出力 (1行)
    fileLineWriter.printDataLine(sampleFileLineObject);

  }

  // トレイラ部の出力
  fileLineWriter.printTrailerLine(trailerString);
  ...
} finally {
  // ファイルのクローズ
  fileLineWriter.closeFile();
}
...
```

ファイル名とパラメータクラスを引数に、
ファイル出力用行ライタを取得する。

ヘッダ部を出力する。
String 型の変数を引数とする。

ファイル形式に関わらず、printDataLine メソッドで出力する。
出力される項目には、項目定義用のアノテーションを付加しておく。

トレイラ部を出力する。
String 型の変数を引数とする。

出力が終了したら、ファイルを
クローズする。

- ファイル出力処理の実装（ファイル出力を行うクラスでファイルをオープンしたまま、追記する場合）

【ジョブ Bean 定義ファイルの設定例】

```
<bean id="blogic" class="testBlogic">
  <property name="writer">
    <bean class="jp.terasoluna.fw.file.dao.standard.
      CSVFileLineWriter"
      destroy-method="closeFile">
      <constructor-arg index="0" value="some_file_path/uriage.csv" />
      <constructor-arg index="1"
        value="jp.terasoluna.batch.sample.SampleFileLineObject" />
      <constructor-arg index="2" ref="columnFormatterMap" />
    </bean>
  </property>
</bean>
```

bean のコンストラクタにファイル名(1 番目の引数。文字型)、パラメータクラス(2 番目の引数、クラス型)、テキスト変換処理(3 番目の引数。FileAccessBean.xml で定義されている Bean "columnFormatterMap"を固定で指定)を設定する。

- ファイル出力処理の実装についての補足
ファイル出力処理を行う場合は、FileUpdateDAO を利用してもファイル出力用行ライタを直接設定する利用する方法のどちらかを利用することで実装出来る。FileUpdateDAO を利用する場合は DAO が呼ばれるたびにファイルのオープン/クローズ処理が行われる。そのため、ファイル出力用行ライタを直接設定した場合と比べて処理が遅くなる。また、ファイル出力用行ライタを直接設定した場合に、ビジネスロジックで出力処理を行い後処理でファイルの移動などを行うと後処理ではまだファイルストリームが存在するため、エラーが発生することに留意すること。
- ファイルの出力順序についての補足
ヘッダ部の出力は、データ部の出力の前に行う必要がある点に留意すること。同様にトレイラ部の出力は、データ部の出力がすべて終わった後に行う必要がある点に留意すること。但し、FileUpdateDAO を前処理、主処理、後処理で利用する場合は順序性が担保出来ているか判断することが出来ないため「トレイラ(前処理)⇒データ(主処理)⇒ヘッダ(後処理)」という出力が不可能となる。前処理、主処理、後処理の各々で FileUpdateDAO を利用する場合はヘッダ部、データ部、トレイラ部の出力処理順序の注意が必要である。

◆ 拡張ポイント

- なし

■ 関連機能

- 『BD-02 対象データ取得機能』
- 『BE-04 リスタート機能』
- 『BH-01 例外ハンドリング機能』

■ 使用例

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

■ 備考

- `encloseChar` で囲い文字を指定しており、囲み文字の前後にデータが存在する場合のファイルアクセス機能の挙動について。

➤ 以下の1行を読み込む場合を想定する。

```
"001",A"toyosu"B
```

➤ ファイル行オブジェクトの実装例(getter/setter は省略)

```
@FileFormat(encloseChar = '"')
public class SampleFileLineObject {
    .....
    @InputFileColumn(columnIndex = 0)
    private String id = null;

    @InputFileColumn(columnIndex = 1)
    private String name = null;
}
```

- このように、囲い文字の前後にデータが存在するような場合、デフォルトではファイル読み込み時に例外「`FileLineException`」をスローする。
- 次ページでこの設定を変更するための手順を以下の2つの例に分けて紹介する。
 1. 囲い文字の中のデータのみ取得し、例外をスローさせない方法。
 2. カラム内のデータ全てを取得し、例外をスローさせない方法。

1. 囲い文字の中のデータを自動的に取得し、例外をスローさせない方法。
(以下のようにファイル行オブジェクトを取得し、処理を継続するパターン)

◇ 取得できるファイル行オブジェクトの内容

id = 001

name = toyosu

- この場合はファイルアクセス用 DAO の Bean 定義において、プロパティ「throwExceptionAtFormatViolation」を false に設定する。
- 以下に、Bean 定義ファイルの設定例を掲載する。

(略)

```
<!-- CSV ファイルアクセス用 (入力) DAO -->
<bean id="csvFileQueryDAO"
      class="jp.terasoluna.fw.file.dao.standard.CSVFileQueryDAO"
      parent="fileQueryDAO">
  <property name="throwExceptionAtFormatViolation" value="false"/>
</bean>

<!-- 可変長ファイルアクセス用 (入力) DAO -->
<bean id="variableFileQueryDAO"
      class="jp.terasoluna.fw.file.dao.standard.VariableFileQueryDAO"
      parent="fileQueryDAO">
  <property name="throwExceptionAtFormatViolation" value="false"/>
</bean>
```

(略)

◇

2. カラム内のデータ全てを取得し、例外をスローさせない方法。
(以下のようにファイル行オブジェクトを取得し、処理を継続するパターン)

id = 001

name = A”toyosu”B

- この場合はファイル行オブジェクトを修正し、カラム毎に個別に囲い文字を設定する事で実現できる。
- 以下に、ファイル行オブジェクトの実装例を掲載する。

```
@FileFormat()
public class SampleFileLineObject {
    .....
    @InputFileColumn(columnIndex = 0 , columnEncloseChar = '"')
    private String id = null;

    @InputFileColumn(columnIndex = 1 ,
                     columnEncloseChar = Character.MIN_VALUE)
    private String name = null;
    (略)
}
```

対象のカラム全体を取得できるように、
個別に囲い文字を設定する。

BC-02 ファイル操作機能

■ 概要

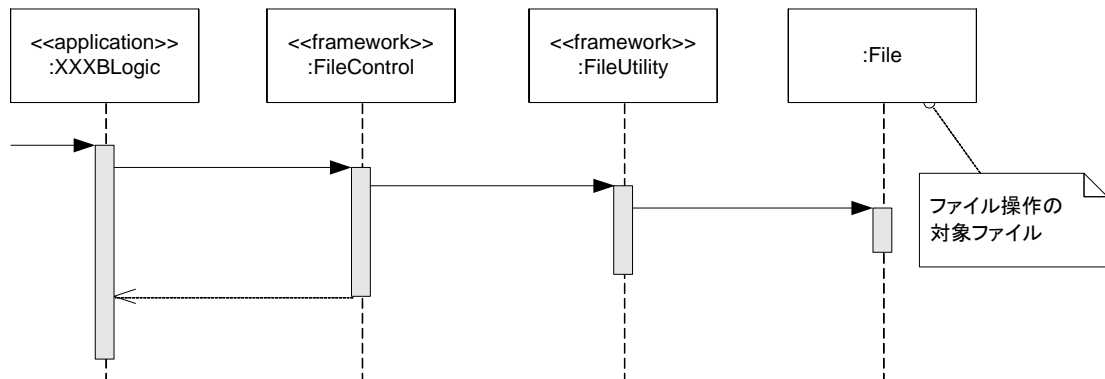
◆ 機能概要

- ファイル操作機能は以下の機能を提供する。
 - ファイル名変更
変更元ファイルと変更先のファイルを指定して、ファイル名の変更を行う。
 - ファイル移動
ファイルの移動は「ファイル名変更」を利用し、パスを変更することにより実現する。
 - ファイルコピー
コピー元ファイルとコピー先ファイルを指定して、ファイルのコピーを行う。
 - ファイル削除
削除するファイルを指定して、ファイルの削除を行う。
 - ファイル結合
指定されたファイル名のリストにあるファイルを結合する。

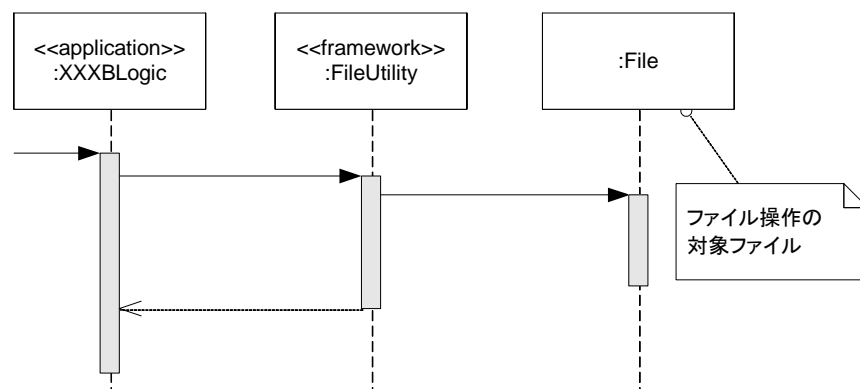
◆ 概念図

ファイル操作の概念図。

- ビジネスロジックからファイル操作インタフェースを利用して、ファイル操作を行なう場合



- ビジネスロジックからファイル操作ユーティリティクラスを直接利用して、ファイル操作を行なう場合



◆ 解説

● FileControl

ファイル操作の処理を提供するインタフェース。インタフェースが提供するメソッドの詳細についてはフレームワークの **JavaDoc** を参照のこと。ここでは、**FileControl** インタフェースで提供するファイル操作の代表的なメソッドを下記に挙げる。

➤ ファイル操作インタフェース

項番	インタフェース名	概要
1	jp.terasoluna.fw.file.util.FileControl	ファイル操作用のインタフェース

➤ ファイル操作インタフェースで提供する代表的なメソッド

項番	メソッド	概要
1	renameFile(String scrFile, String newFile)	ファイル名の変更・ファイルの移動
2	copyFile(String scrFile, String newFile)	ファイルのコピー
3	deleteFile(String scrFile)	ファイルの削除
4	mergeFile(List<String> fileList, String newFile)	fileList にあるファイルの結合

● FileControlImpl

FileControl インタフェースを実装するクラス。**FileControlImpl** クラスはビジネスロジックに **DI** して利用することができる。**FileControlImpl** クラスはファイル操作処理を実行する **FileUtility** クラスをラップしている。

FileControlImpl クラスは、属性にファイル操作を行う際に基準となるパス（基準パス）を持つ。基準パスを使用することにより、ファイルアクセス時に発生するファイルパスの環境依存の問題を回避することができる。ファイル操作を行うメソッドの引数には相対パス、または、絶対パスを設定する。ファイル操作時にエラーが発生した場合、非検査例外をスローする。

➤ 基準パスはファイル操作機能を使う上での基準となる位置を指す。基準パスを「/si1/」、相対パスを「chohyo/test.txt」とした場合、ファイルの絶対パスは「/si1/chohyo/test.txt」となる。

- FileUtility

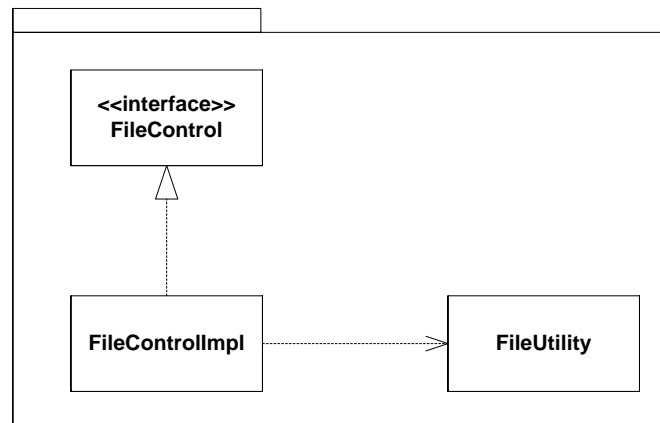
ファイル操作機能を実装するクラス。FileUtility のメソッドをビジネスロジックから直接利用することも可能である。FileUtility クラスで提供するメソッドは FileControl インタフェースで提供するものと同じになる。FileUtility クラスではファイル操作を行うメソッドの引数は絶対パスのみ設定可能である。ファイル操作時にエラーが発生した場合、非検査例外をスローする。

- ファイル操作ユーティリティクラス

項番	メソッド	概要
1	jp.terasoluna.fw.file.util.FileUtility	ファイル操作ユーティリティクラス

- ファイル操作機能のクラス図

- FileControl、FileControlImpl、FileUtility クラスの関連は下記のクラス図のとおりである。



■ 使用方法

◆ コーディングポイント

- ファイル操作クラス（FileControl）を利用する例

➤ Bean 定義ファイルの設定例

```

.....
<bean id="fileControl"
      class="jp.terasoluna.fw.file.util.FileControlImpl">
  <property name="basePath" value="${basepath}" />
  <property name="checkFileExist" value="false" />
</bean>
.....
<bean id="sampleBLogic"
      class="jp.terasoluna.batch.sample.SampleBLogic">
.....
  <property name="fileControl" ref="fileControl" />
.....

```

FileControl インタフェースを実装するクラスをフレームワーク Bean 定義ファイルに定義する。プロパティに基準パスを設定すること。

操作後にできるファイルパスにファイルが存在する場合、処理を継続する(上書きする: true)か例外を投げて停止する(false)かを定めるフラグ。

フレームワーク Bean 定義ファイルで設定した Bean を参照する

➤ ビジネスロジックの実装例（ファイルのコピー、移動、削除処理の実装例）

```

.....
private FileControl fileControl = null;
public void setFileControl(FileControl fileControl){
  this.fileControl = fileControl;
}
.....
// ファイルのコピー（相対パスを設定する例）
// /sil/chohyo/test.txt を/sil/chohyo/testFile.txt にコピー。
// 基準パスは「/sil/」
fileControl.copyFile("chohyo/test.txt", "chohyo/testFile.txt");
.....
// ファイルの移動（相対パスを設定する例）
// /sil/chohyo/testFile.txt を/sil/output/testFile.txt に移動。
// 基準パスは「/sil/」
fileControl.renameFile("chohyo/testFile.txt", "output/testFile.txt");
.....
// ファイルの削除（相対パスを設定する例）
// /sil/chohyo/testFile.txt を削除。
// 基準パスは「/sil/」
fileControl.deleteFile("chohyo/testFile.txt");
.....
// ファイルのコピー（絶対パスを設定する例）
// /sil/chohyo/test.txt を/sil/chohyo/testFile.txt にコピー。
fileControl.copyFile("/sil/chohyo/test.txt", "/sil/chohyo/testFile.txt");
.....

```

ファイル操作機能を利用するクラスは、FileControl インタフェースとその setter が必須

各メソッドの引数はファイルの相対パス、もしくは絶対パスを記述する

➤ ビジネスロジックの実装例（ファイル結合の実装例）

```
.....
// ファイルの結合。
// 以下に挙げるファイルをリストに格納し、ファイルを/sil/output/mergeFile.csvに統合。
// /sil/chohyo/output001.csv
// /sil/chohyo/output002.csv
// /sil/chohyo/output003.csv
// 基準パスは「/sil/」
fileList.add("chohyo/output001.csv");
fileList.add("chohyo/output002.csv");
fileList.add("chohyo/output003.csv");
.....
fileControl.mergeFile(fileList, "output/mergeFile.csv");
.....
```

メソッドの2番目の引数はファイルの相対パス、もしくは絶対パスを記述する

● ファイル操作ユーティリティクラス（FileUtility）を直接利用する例

➤ ビジネスロジックの実装例（ファイルのコピー、移動、削除処理の実装例）

```
.....
// ファイルのコピー。
// /sil/chohyo/test.txt を/sil/chohyo/testFile.txtにコピー。
FileUtility.copyFile("/sil/chohyo/test.txt", "/sil/chohyo/testFile.txt");
.....
// ファイルの移動。
// /sil/chohyo/testFile.txt を/sil/output/testFile.txtに移動。
FileUtility.renameFile("/sil/chohyo/testFile.txt",
"/sil/output/testFile.txt");
.....
//ファイルの削除。 /sil/chohyo/testFile.txt を削除。
FileUtility.deleteFile("/sil/chohyo/testFile.txt");
.....
```

各メソッドの引数はファイルの絶対パスを記述する

◆ 拡張ポイント

- なし。

■ 関連機能

- なし。

■ 使用例

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

■ 備考

- なし。

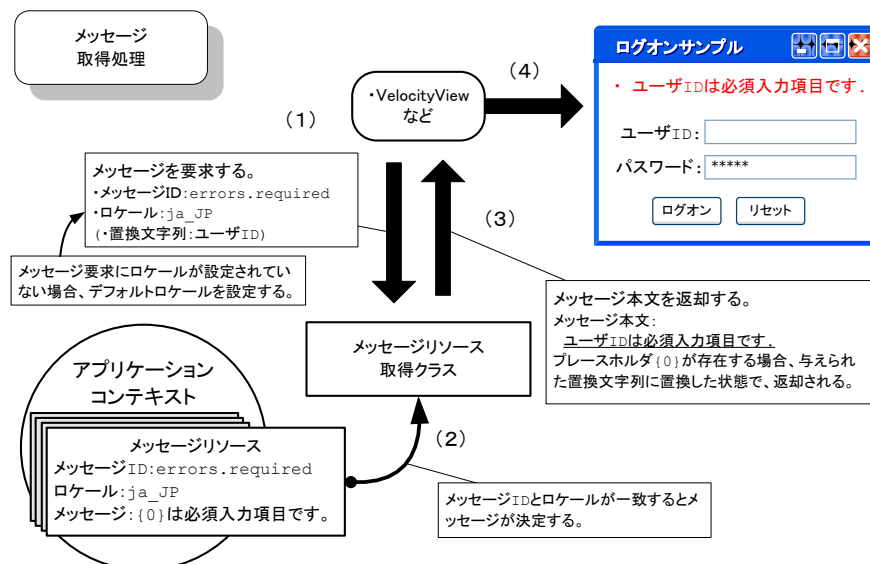
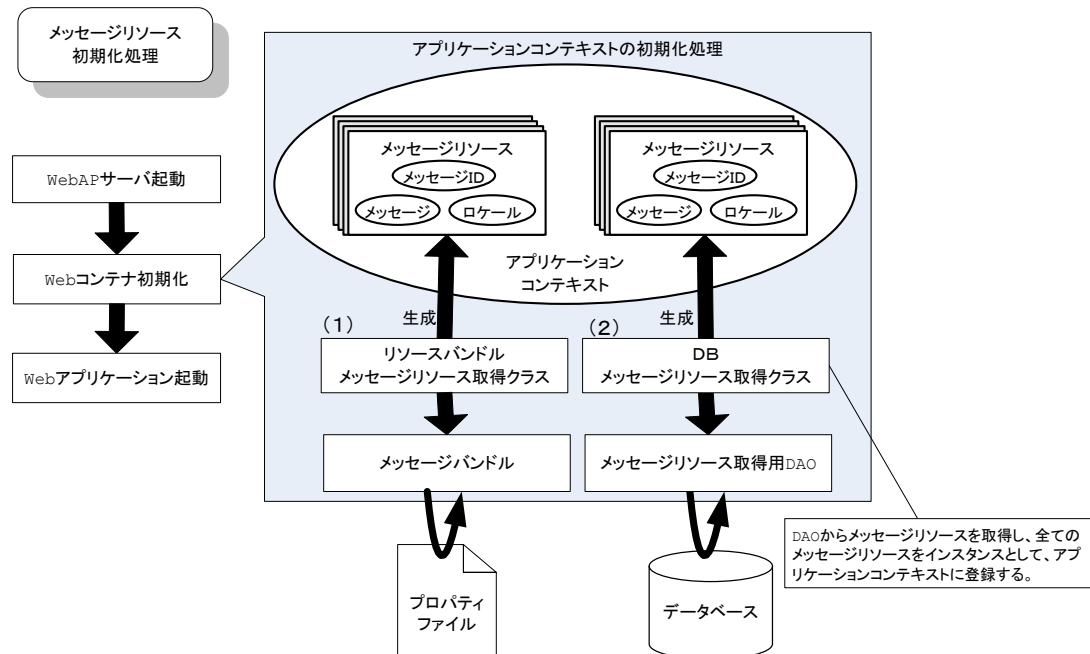
CE-01 メッセージ管理機能

■ 概要

◆ 機能概要

- アプリケーションユーザなどに対して表示する文字列(メッセージリソース)を、定義できる。
- メッセージリソースは、プロパティファイルやデータベース内のメッセージ定義テーブルに定義することができる。
- 国際化に対応しており、ユーザのロケールに応じたメッセージを定義できる。

◆ 概念図



◆ 解説

- メッセージリソース初期化处理

- (1) リソースバンドルメッセージリソース
リソースバンドルを利用してプロパティファイルを読み込み、アプリケーションコンテキストに保持する。
- (2) DBメッセージソース
メッセージリソース取得用 DAO を用いて、データベース内に定義された全てのメッセージを取り出し、{メッセージ ID, ロケール, メッセージ文字列} の組としてアプリケーションコンテキストに保持する。

- メッセージリソース取得処理

- (1) メッセージを要求する
取得したいメッセージのメッセージ ID およびロケール文字列を引数に指定して呼び出す。
- (2) メッセージリソースを検索する
アプリケーションコンテキストのメッセージリソース内を検索して該当するメッセージを取得する。
- (3) メッセージを返却する
取得したメッセージを要求元に返却する。
- (4) メッセージを利用する
View やビジネスロジックなどでメッセージを利用する。

※メッセージリソース中にプレースホルダ(概念図中の「{0}」)を定義しておくことで、引数に指定した文字列を動的に埋め込んだメッセージを取り出すことができる。

■ 使用方法

◆ コーディングポイント

- ソフトウェアアーキテクトが行うコーディングポイント（リソースバンドル）
以下のように“messageSource”という識別子の Bean を準備することで、この機能を利用できる。

➤ Bean 定義ファイルサンプル（applicationContext.xml）

```
<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames"
            value="application-messages, system-messages"/>
</bean>
```

messageSource を指定する。

読み込むプロパティファイルをカンマ区切りで列挙する。ファイル名の“.properties”は省略する。

プロパティファイルはクラスパス上に配置する。

定義するプロパティファイルが多い場合は、下記のようにリストの形で指定することもできる。

➤ Bean 定義ファイルサンプル（applicationContext.xml）

```
<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames">
    <list>
      <value> application-messages </value>
      <value> system-messages </value>
    </list>
  </property>
</bean>
```

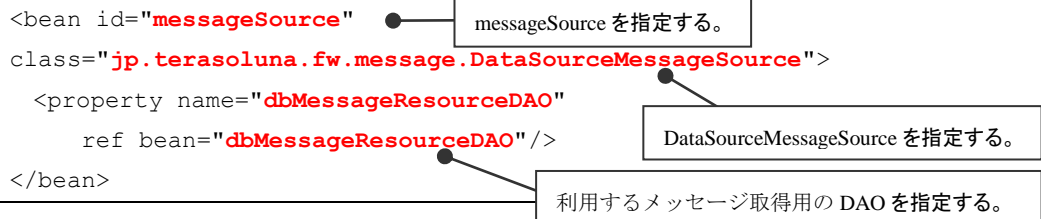

- ソフトウェアアーキテクトが行うコーディングポイント（DBメッセージ）
以下のように “messageSource” という識別子の Bean を準備することで、この機能を利用できる。

➤ メッセージリソース取得 Bean の定義

TERASOLUNA Server Framework for Java が提供している

DataSourceMessageSource クラスを指定し、DAO(後述)を DI する。BeanID は “messageSource” である必要がある。

◇ Bean 定義ファイルサンプル (applicationContext.xml)

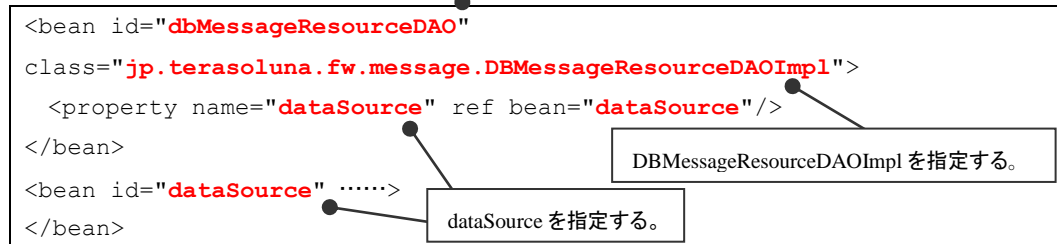


➤ メッセージリソース取得用 DAO の Bean 定義

DBMessageResourcesDAO インタフェースを指定し、データソースを DI する。

TERASOLUNA Server Framework for Java ではこのインタフェースのデフォルト実装として DBMessageResourceDAOImpl クラスを提供している。

◇ Bean 定義ファイルサンプル (dataAccessContext-local.xml)



➤ データソースの定義

『CB-01 データベースアクセス機能』を参照して設定する。

➤ メッセージ文字列の定義

メッセージ文字列はデータベース中の以下のテーブルに格納しておく：

- ・テーブル名 : MESSAGES
- ・メッセージコードを格納するカラム名 : CODE
- ・メッセージ本文を格納するカラム名 : MESSAGE

DBMessageResourceDAOImpl が発行する SQL は以下である。

SELECT CODE,MESSAGE FROM MESSAGES

テーブルスキーマを自由に定義することも可能である。後述「メッセージリソースのテーブルスキーマをデフォルト値から変更する」を参照されたい。

- 業務開発者が行うコーディングポイント
 - メッセージの取得方法
 - ◇ 例外メッセージの取得 (Rich 版の場合)

Velocity ビューを利用することで、例外メッセージの取得をコーディングレスで行える。詳細は『RB-02 レスポンスデータ生成機能』の説明書を参照のこと。
 - ◇ その他、正常系メッセージなどの取得

DI コンテナで管理するクラスが、上記、『例外ハンドリング機能』を利用せずにメッセージを取得する場合は、以下のクラスを利用する。

org.springframework.context.support ApplicationObjectSupport

上記クラスで定義されている `MessageSourceAccessor` 内の `getMessage` メソッドを使用する。詳細については `MessageSourceAccessor` の `JavaDoc` を参照のこと。各ビジネスロジックが直接、`getMessage` メソッドを使用することはせず、メッセージ取得用クラスなどの業務共通クラスから利用することを推奨する。

- メッセージ使用例

- メッセージ取得クラスインタフェースサンプル

```
public interface MessageAccessor {  
    //メッセージをそのまま取り出す場合  
    public String getMessage(String code, Object[] args);  
    ...省略...  
}
```

業務共通機能担当者が作成する。

ビジネスロジック開発者が使用するメソッドを規定する。

- メッセージ取得クラス実装クラスサンプル

```
public class MessageAccessorImpl extends ApplicationObjectSupport implements  
MessageAccessor {  
    //メッセージをそのまま取り出す場合  
    public String getMessage(String code, Object[] args) {  
        return getMessageSourceAccessor().getMessage(code, args);  
    }  
    ...省略...  
}
```

業務共通機能担当者が作成する。

`ApplicationObjectSupport` クラス内の `MessageSourceAccessor` オブジェクトの `getMessage` メソッドを利用する。

➤ ビジネスロジックサンプル

```
public class SampleBLogic implements BLogic {
```

```
    //メッセージ出力クラス用setter
```

```
    MessageAccessor msgAcc = null;
```

```
    public void setMsgAcc(MessageAccessor msgAcc) {
```

```
        this.msgAcc = msgAcc;
```

```
    }
```

```
    //ビジネスロジック
```

```
    public ResultBean sampleLogic(String teamId) throws Exception {
```

```
        ResultBean result = new ResultBean();
```

```
        String outPutMessage = null;
```

```
        outPutMessage = msgAcc.getMessage("welcome", teamId);
```

```
        result.setResult(outPutMessage, .....(省略).....);
```

```
        return result;
```

```
    }
```

```
}
```

ビジネスロジック開発担当者が作成する。

上記、メッセージ出力クラスを DI するためのSetterを記述する。

メッセージ出力クラスからメッセージ取得メソッドを利用する。

➤ Bean 定義ファイルサンプル(applicationContext.xml)

```
<!--メッセージ出力クラス -->
```

```
<bean id="msgAcc" class="jp.terasoluna.fw.message.MessageAccessorImpl"/>
```

```
<!-- 業務ロジッククラス -->
```

```
<bean id="sampleBLogic" class="jp.terasoluna.sample.service.impl.SampleBLogic">
```

```
    <property name="messageAccessor" ref="msgAcc"/>
```

```
</bean>
```

業務共通機能担当者が記述する。

ビジネスロジック開発担当者が記述する。

➤ メッセージリソースの再定義方法

Web アプリケーション起動中にアプリケーションコンテキスト内のメッセージをデータベースから再取得することができる。再定義には以下のメソッドを使用する。なお、クラスタ環境化では、クラスタごとに再定義する必要があるので注意されたい。以下のメソッドを使用する。

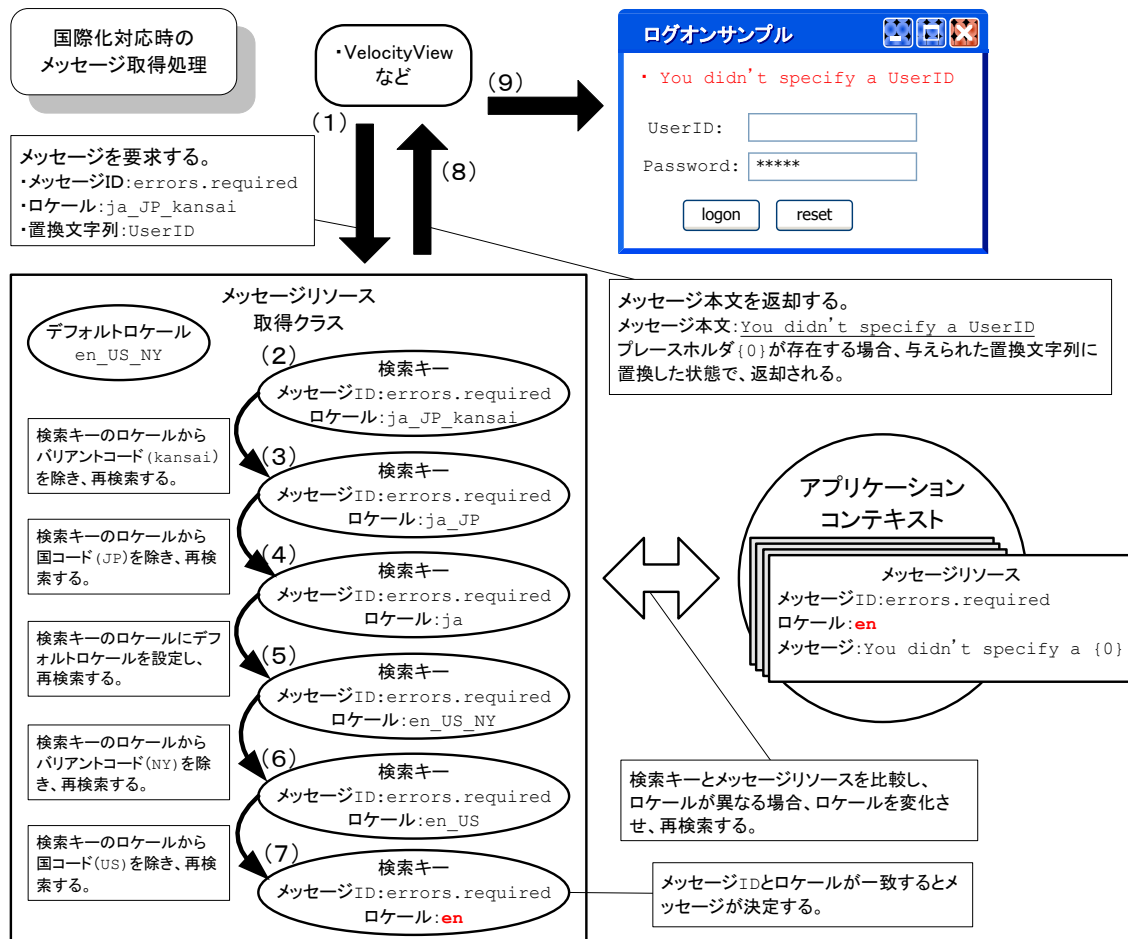
**jp.terasoluna.fw.message.DataSourceMessageSource クラスの
reloadDataSourceMessage メソッド**

各ビジネスロジックが直接、reloadDataSourceMessage メソッドを使用することとはせず、業務共通クラスから利用することを推奨する。

■ メッセージの国際化対応

◆ 国際化対応時のメッセージ決定ロジック

● 概要図



● 解説

- (1) 取得したいメッセージのメッセージ ID、ロケールを検索キーとして、また必要な場合は置換文字列を引数として渡す。なお、ロケールはクライアントのリクエストから取得する。取得出来ない場合はサーバー側で設定されているデフォルトロケールが設定される。
- (2) 与えられたメッセージ ID とロケールを検索キーとし、メッセージの検索をする。
- (3) (2)でメッセージが決定されず、検索キーのロケールにバリエーションコードがある場合、バリエーションコードを除き、再検索する。
- (4) (3)でメッセージが決定されず、検索キーのロケールに国コードがある場合、国コードを除き、再検索する。
- (5) (4)でメッセージが決定されない場合は、検索キーのロケールにデフォルトロケールを設定し、再検索する。
- (6) (5)でメッセージが決定されず、検索キーのロケールにバリエーションコードがある場合、

バリエーションコードを除き、再検索する。

- (7) (6)でメッセージが決定されず、検索キーのロケールに国コードがある場合、国コードを除き、再検索する。
- (8) 決定されたメッセージを返却する。決定されたメッセージにプレースホルダ（概念図では {0}）が存在する場合（**MessageFormat** 型）は引数として渡された置換文字列に置き換える。
- (9) 取得したメッセージを用い、画面に表示する。

- ソフトウェアアーキテクトが行うコーディングポイント
 - デフォルトロケールの設定
メッセージリクエストにロケールが設定されていない場合、及びメッセージリソース内にメッセージリクエストで要求されたロケールが見つからない場合に使用される。設定しない場合はデフォルトロケールの初期設定（サーバー側 VM のロケール）が使用される。

- Bean 定義ファイルサンプル（applicationContext.xml）

```
<bean id="messageSource"
      class="jp.terasoluna.fw.message.DataSourceMessageSource">
  <property name="dbMessageResourceDAO"
    ref bean="dbMessageResourceDAO"/>
  <property name="defaultLocale" value="ja_JP"/>
</bean>
```

デフォルトロケールを指定する。

- 国際化対応カラムの有効化
データベースのロケールに対応するカラムからの読み込みを有効にする必要がある。ロケールに対応するカラムは以下の3つがある。

- ・ 言語コードカラム
- ・ 国コードカラム
- ・ バリエーションコードカラム

設定の優先順位は、言語コードカラムが一番高く、国コードカラム、バリエーションコードカラムの順に低くなる。言語コードカラムを指定せずに、国コードカラムやバリエーションコードカラムを指定しても無効となる。

これらのカラムのうち、言語コードカラムの指定によってデータベースに登録されたメッセージの認識が以下のように変化する。

- ・ **言語コードカラムを指定しない場合は**、すべてのメッセージがデフォルトロケールとして認識される。（defaultLocale プロパティを指定した場合はその値となる）
- ・ **言語コードカラムを指定した場合は**、言語コードカラムに指定したとおりに認識される。

注意点としては、言語コードカラムを指定し、言語コードカラムに null や空文字のメッセージをデータベースに登録した場合、そのメッセージはアプリケーションから参照されない点である。null や空文字で登録したメッセージがデフォルトロケールとして認識されるわけではない点に注意。

以下のプロパティで設定されていない値はデフォルトの値が使用される。設定する項目は以下の通り。

プロパティ名	デフォルト値	概要	備考
languageColumn	null	言語コードを格納するカラム名	国際化対応時のみ設定
countryColumn	null	国コードを格納するカラム名	国際化対応時のみ設定
variantColumn	null	バリエントコードを格納するカラム名	国際化対応時のみ設定

メッセージ取得 SQL 文のフォーマットは以下の通り。

SELECT メッセージコード, (言語コード), (国コード), (バリエントコード), メッセージ本体 **FROM** テーブル名 **FROM** テーブル名

()内は設定した値のみが有効になる。デフォルトでは無効になっており、カラム名を設定すると有効になる。

● Bean 定義ファイルサンプル (dataAccessContext-local.xml)

```
<bean id=DBMessageResourceDAO
  class="jp.terasoluna.fw.message.DBMessageResourceDAOImpl">
  <property name="dataSource" ref bean="dataSource"/>
  <property name=tableName value="DBMESSAGES"/>
  <property name=codeColumn value="BANGOU"/>
  <property name=languageColumn value="GENGO"/>
  <property name=countryColumn value="KUNI"/>
  <property name=variantColumn value="HOUGEN"/>
  <property name=messageColumn value="HONBUN"/>
</bean>
```

国際化対応する場合のみ設定。
言語コードのカラム名を指定する。

国際化対応する場合のみ設定。
国コードのカラム名を指定する。

国際化対応する場合のみ設定。
バリエントコードのカラム名を指定する。

DBのテーブル名及びカラム名は以下の様な設定となる。

テーブル名 = DBMESSAGES

メッセージコードを格納するカラム名 = BANGOU

メッセージの言語コードを格納するカラム名 = GENGO

メッセージの国コードを格納するカラム名 = KUNI

メッセージのバリエントコードを格納するカラム名 = HOUGEN

メッセージ本文を格納するカラム名 = HONBUN

検索SQL文は以下の通り。

SELECT BANGOU,GENGO,KUNI,HOUGEN,HONBUN FROM DBMESSAGES

■ リファレンス

◆ 構成クラス

	クラス名	概要
1	DataSourceMessageSource	メッセージを生成、発行するクラス
2	DBMessageResourceDAOImpl	DB よりメッセージリソースを抽出する DBMessageResourceDAO の実装クラス
3	MessageSource	メッセージ取得のためのメソッドを規定したインタフェイスクラス

◆ 拡張ポイント

- なし

■ 関連機能

- なし

■ 使用例

- Terasoluna Server Framework for Java (Web 版) 機能網羅サンプル
 - WG-01 メッセージ管理機能
- Terasoluna Server Framework for Java (Rich 版) 機能網羅サンプル
 - UC110 DB メッセージ管理
 - ✧ `jp.terasoluna.rich.functionsample.dbmessage.*`

■ 備考

◆ メッセージリソースのテーブルスキーマをデフォルト値から変更する

- テーブル名、カラム名をプロジェクト側で独自に指定する場合
テーブル名及び各カラム名のすべてもしくは一部を設定することでデータベースのテーブル名及びカラム名を自由に変更できる。設定されていない値はデフォルトの値が使用される。設定する項目は以下の通り。

プロパティ名	デフォルト値	概要
tableName	MESSAGES	テーブル名
codeColumnn	CODE	メッセージコードを格納するカラム名
messageColumnn	MESSAGE	メッセージ本文を格納するカラム名

メッセージ取得 SQL 文の SELECT 節のフォーマットは以下の通り。

SELECT メッセージコード, メッセージ本体

なお、この設定では国際化に未対応となる。国際化対応が必要な場合は、前述の『メッセージの国際化対応』の項目を参照のこと。

例) データベースのテーブル名及びカラム名を以下の様にする場合

- ・ テーブル名 : DBMESSAGES
- ・ メッセージコードを格納するカラム名 : BANGOU
- ・ メッセージ本文を格納するカラム名 : HONBUN

➤ Bean 定義ファイルサンプル (dataAccessContext-local.xml)

```
<bean id="dbMessageResourceDAO"
class="jp.terasoluna.fw.message.DBMessageResourceDAOImpl">
  <property name="dataSource" ref="dataSource"/>
  <property name="tableName" value="DBMESSAGES"/>
  <property name="codeColumnn" value="BANGOU"/>
  <property name="messageColumnn" value="HONBUN"/>
</bean>
```

テーブル名を指定する。

メッセージコードのカラム名を指定する。

メッセージ本文のカラム名を指定する。

メッセージ取得 SQL 文は以下の通り。

SELECT BANGOU,HONBUN FROM DBMESSAGES

- 上記『テーブル名、カラム名をプロジェクト側で独自に指定する場合』に加え、プロジェクト独自の SQL 文を設定する場合
findMessageSql プロパティで独自の SQL 文を設定できる。設定する SQL 文には、

codeColumn プロパティおよび messageColumn で指定したカラムが必要となる。
設定する項目は以下の通り。

プロパティ名	デフォルト値	概要
tableName	MESSAGES	テーブル名
codeColumn	CODE	メッセージコードを格納するカラム名
messageColumn	MESSAGE	メッセージ本文を格納するカラム名
findMessageSql	-	メッセージを取得する SQL 文

例) メッセージ取得 SQL 文を『SELECT BANGOU,HONBUN FROM DBMESSAGE WHERE CATEGORY='TERASOLUNA'』とする場合。

- ・テーブル名 : DBMESSAGES
- ・メッセージコードを格納するカラム名 : BANGOU
- ・メッセージ本文を格納するカラム名 : HONBUN

➤ Bean 定義ファイルサンプル (dataAccessContext-local.xml)

```
<bean id="dbMessageResourceDAO"
    class="jp.terasoluna.fw.message.DBMessageResourceDAOImpl">
    <property name="dataSource" ref="dataSource"/>
    <property name="tableName" value="DBMESSAGES"/>
    <property name="codeColumn" value="BANGOU"/>
    <property name="messageColumn" value="HONBUN"/>
    <property name="findMessageSql"
        value=
        "SELECT BANGOU,HONBUN FROM DBMESSAGE WHERE CATEGORY='TERASOLUNA'"
    />
</bean>
```

テーブル名を指定する。

メッセージコードのカラム名を指定する。

メッセージ本文のカラム名を指定する。

検索 SQL 文を指定する。

◆ 第2メッセージリソースの使用

メッセージリソースを追加できる。追加したメッセージリソースは前述で "messageSource" という識別子の Bean として設定したメッセージリソースでメッセージが決定できない場合に利用される。以下のように "parentMessageSource プロパティ" に別のメッセージリソースへの参照を指定することで、この機能を利用できる。

☆ 第2メッセージリソース取得 Bean の定義

利用したいメッセージリソース取得クラスを指定する。BeanID は "messageSource" とは別の名前を付与する必要がある。

AbstractMessageSource の継承クラスであれば、この "parentMessageSource プロパティ" を利用できるので、さらに第3、4 とリンクすることが可能である。

➤ Bean 定義ファイルサンプル (applicationContext.xml)

```
<bean id="messageSource"
      class="jp.terasoluna.fw.message.DataSourceMessageSource">
  <property name="parentMessageSource" ref="secondMessageSource"/>
  <property name="dbMessageResourceDAO" ref="dbMessageResourceDAO"/>
</bean>

<bean id="secondMessageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames" value="applicationResources,errors"/>
</bean>
```

messageSource を指定する。優先して検索されるメッセージリソースとなる。

次に参照される

第2のメッセージリソースを指定する。上記 messageSource 内にメッセージが存在しなかった場合、ここで指定したメッセージリソース内を検索する。