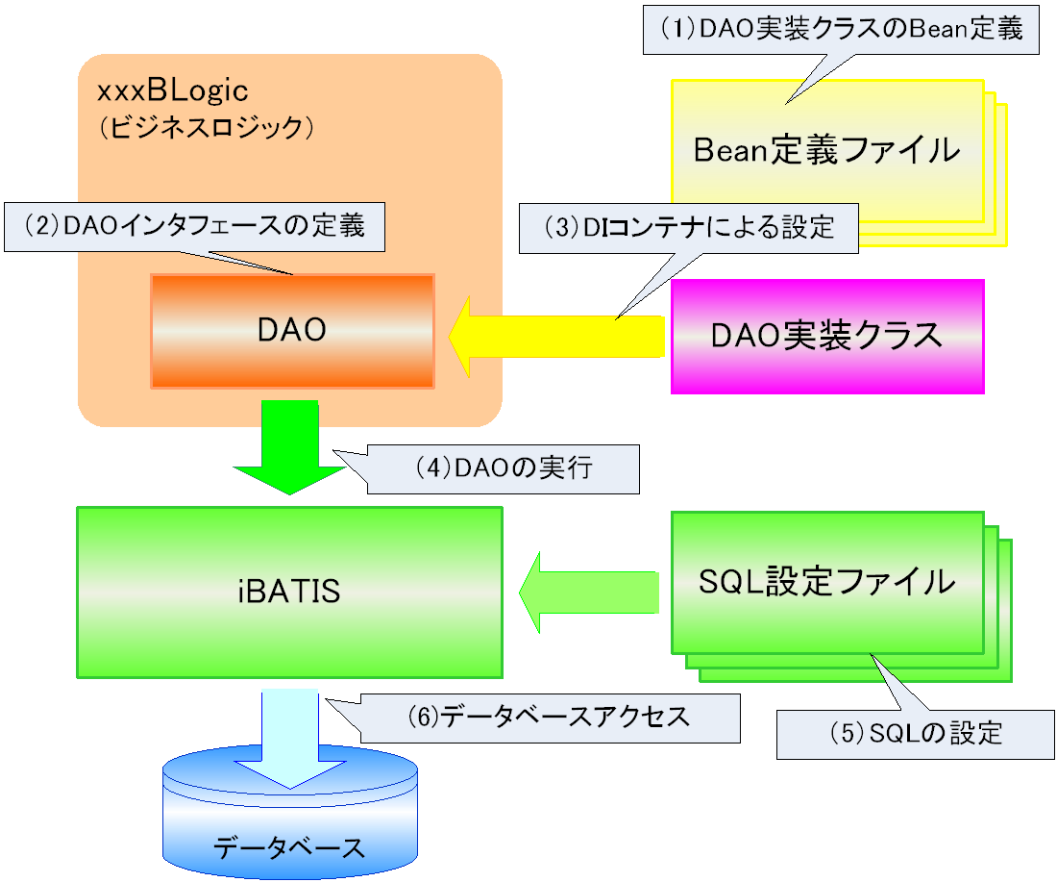
CB-01 データベースアクセス機能

■ 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.xsd
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

<!-- 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>
```

Property-placeholder を定義する。

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.xsd
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

  <!-- JDBC関連のプロパティ -->
  <context:property-placeholder location="WEB-INF/jdbc.properties"/>

  <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>
```

Property-placeholder を定義する。

スキーマを定義する。

クラスパスを指定する。

DriverManagerDataSource の定義。

※ 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”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.executeForArray("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を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.executeForList("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ーを実行する。

  - iBATIS マッピング定義ファイル
    - プロシージャー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ー（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。