AxiZ
Academy eXtend IT skill Zero

実践 Spring Framework + MyBatis

MyBatis – Object/Relational Mapping 編

ver. 1.1.1

TC. Techno-core Corporation 株式会社テクノコア
http://www.techno-core.jp

はじめに

ORM(Object/Relational Mapping)フレームワークとは?

Webアプリケーション開発において、DBとの連携処理は重要な要素です。
つまり、SQLによるRDBMSの利用は、サーバサイドJavaの開発においてほぼ必須の知識となっています。

そして、開発時またはメンテナンス時に常に問題となり、バグの温床となり易いのが、「DBアクセス」や「SQL」、「抽出データのオブジェクト変換」処理です。

DBスキーマは不変ではありません。
開発中はもちろん、リリース後でも仕様変更やエンハンスにより、常に変更されていきます。
そのため、ほんの些細なデータ構造の変更が、煩雑なソースコードの修正作業を繰り返し発生させることでしょう。

このような問題を解決するため、開発プロジェクトでは「DBアクセス独自フレームワーク」を用意することがあり、このフレームワークによって、DBアクセス処理の方式が統一され、メンテナンス性の向上が見込まれます。

しかし、それにより今度は「独自フレームワークの開発コスト」や、「PGのフレームワーク習得コスト」の問題が生まれてしまいます。

その解として採用されるのが、HibernateやMyBatisに代表されるオープンソース「ORM(Object/Relational Mapping)フレームワーク」です。

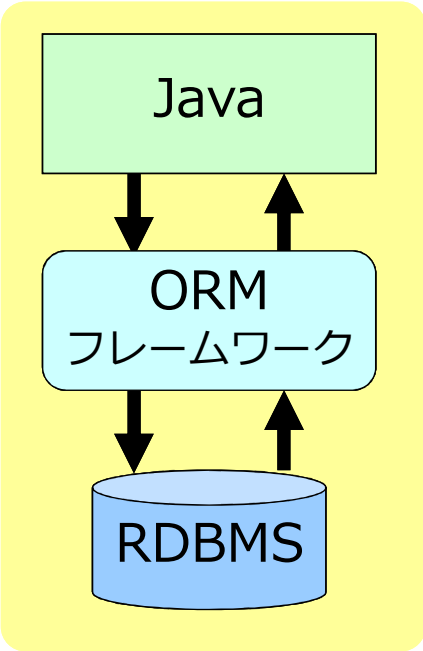
はじめに

ORM(Object/Relational Mapping)

ORM(Object/Relational Mapping)とは、「(Javaの)オブジェクト」と「RDBMSのデータ」をマッピングする技術です。

Javaのプログラムは、ORMフレームワークを経由してRDBMSに対してSQLを発行し、ORMフレームワークはSQLに基づいてRDBMSから抽出したデータを、Javaのプログラムで利用しやすいオブジェクトとしてアプリケーション側に提供します。

つまり、ORMとはJavaのWebアプリケーションとRDBMSの間で、両者の橋渡しの役割を果たします。



ORMの役割

ORMの役割は、主に以下の2点が挙げられます。

1) RDBMS依存コードの排除

ソースコード内から、SQLリテラルやマッピング情報、DBアクセス処理等のRDBMSに依存するコードを排除し、それらは設定ファイルなどで管理します。

2) インピーダンス･ミスマッチの吸収

「オブジェクト指向のオブジェクト」と「RDBMSのテーブル(データ)」間のインピーダンス･ミスマッチ(構造的相違)を解決し、オブジェクトとデータの関連付けを行います。

はじめに

MyBatisの特徴

MyBatisは、前身であるiBATISから移行したORMで、特徴としては、「簡素なAPI」と「SQLとオブジェクトとのマッピング」が挙げられます。

【簡素なAPI】

非常に簡素なAPIでDBアクセスを実現することができ、それによりソースコードの可読性やメンテナンスの向上、技術習得コストの軽減が見込めます。

【SQLとオブジェクトとのマッピング】

「オブジェクトとテーブルをマッピングする」というよりも、「オブジェクトとSQLをマッピングする」というイメージに近いです。

MyBatisでは、プログラマが直接SQLを記述するため、発行するSQLを完全にコントロールすることができ、柔軟なクエリやパフォーマンスチューニングをし易いORMフレームワークとなっています。

☆今回の開発環境は、以下の構成で構築します。

【RDBMS】
MySQL5.5.27
【ORM】
MyBatis3.2.2

開発環境構築(MySQL)

MySQL DBの作成

rootでMySQLにログインし、今回使用するDBとユーザを作成します。
Shellを起動しMySQLにログインします。

```
C:¥>mysql -u root -p   // rootでログイン.
Enter password:
Welcome to the MySQL monitor.
 .....
mysql>                 // ログイン成功.
```

次にDBを作成します。

```
mysql>CREATE DATABASE axiz_db;      // 作成するDBのID = axiz_db.
Query OK, 1 row affected (0.02 sec) // axiz_db作成成功.
mysql>
```

開発環境構築(MySQL)

MySQL ユーザの作成

axiz_db接続ユーザを作成します。

```
mysql>CREATE USER 'axizuser'@'localhost' IDENTIFIED BY 'axiz'; // 作成するユーザのID = axizuser, PWD = axiz.
Query OK, 0 row affected (0.09 sec)                            // axizuser作成成功.
mysql>
```

最後に、作成したaxizuserに対してaxiz_db全テーブルの操作権限を付与します。

```
mysql>GRANT ALL ON axiz_db.* TO 'axizuser'@'localhost'; // axiz_dbのテーブルに対する全権限を付与.
Query OK, 0 row affected (0.00 sec)                     // axizuser作成成功.
mysql>
mysql>FLUSH PRIVILEGES;                                 // 権限変更の反映.
Query OK, 0 row affected (0.00 sec)
mysql>
```

開発環境構築(MySQL)

MySQL ユーザの作成

一旦rootはログアウトし、作成したaxizuserでログイン後axiz_dbに接続できれば成功です。

```
mysql>¥q                    // ログアウト.
Bye                         // ログアウト完了.
C:¥>
C:¥>mysql -u axizuser -p    // axizuserでログイン.
Enter password: ****        // PWD = axiz.
Welcome to the MySQL monitor.
 .....
mysql>                      // ログイン成功.
mysql>¥r axiz_db       // axiz_dbに接続.
Connection id:    1
Current database: axiz_db
mysql>                      // 接続成功.
```

開発環境構築(MySQL)

MySQL テーブル/データの作成

ユーザマスタとユーザ情報を格納する、それぞれ簡単なテーブルを作成し、データを投入します。以降のハンズオンでは、このテーブルとデータを使用して進めます。

```
-- WindowsコマンドプロンプトからMySQLに日本語文字データを登録するため.
SET NAMES CP932;

-- ユーザマスタTBL.
CREATE TABLE usr_mst (
    usr_id      VARCHAR(25)  PRIMARY KEY,
    usr_pwd     VARCHAR(25)  NOT NULL,
    is_del_flg  CHAR(1)      NOT NULL
);

-- ユーザ情報TBL.
CREATE TABLE usr_inf (
    usr_id      VARCHAR(25)  PRIMARY KEY,
    usr_name    VARCHAR(25)  NOT NULL,
    usr_tel     VARCHAR(12)      NULL,
    usr_addr    VARCHAR(50)      NULL,
    is_del_flg  CHAR(1)      NOT NULL
);

-- ユーザマスタTBL_INSERT.
INSERT INTO usr_mst (usr_id, usr_pwd, is_del_flg) VALUES ('user', 'u_axiz', '0');
INSERT INTO usr_mst (usr_id, usr_pwd, is_del_flg) VALUES ('admin', 'a_axiz', '0');
```

開発環境構築(MySQL)

MySQL テーブル/データの作成

```
-- ユーザ情報TBL_INSERT.
INSERT INTO usr_inf (usr_id, usr_name, usr_tel, usr_addr, is_del_flg)
VALUES ('user', 'AxiZユーザ', '0366677081', '東京都中央区日本橋本町3丁目8-4第2東硝ビル2F', '0');
INSERT INTO usr_inf (usr_id, usr_name, usr_tel, usr_addr, is_del_flg)
VALUES ('admin', 'AxiZ管理者', '0366677080', '東京都中央区日本橋本町3丁目8-4', '0');
```

開発環境構築(MyBatis)

MyBatis ライブラリ

http://goo.gl/eAYma から mybatis-3.2.2.zip をダウンロードします。
zipファイルを解凍すると、以下の構成で展開されます。

【mybatis-3.2.2.zip】

```
mybatis-3.2.2
  |-- mybatis-3.2.2.jar
  |-- mybatis-3.2.2.pdf
  |-- LICENSE
  |-- NOTICE
  |-- lib
        |-- asm-3.3.1.jar
        |-- cglib-2.2.2.jar
        |-- commons-logging-1.1.1.jar
        |-- log4j-1.2.17.jar
        |-- slf4j-api-1.7.5.jar
        |-- slf4j-log4j12-1.7.5.jar
```

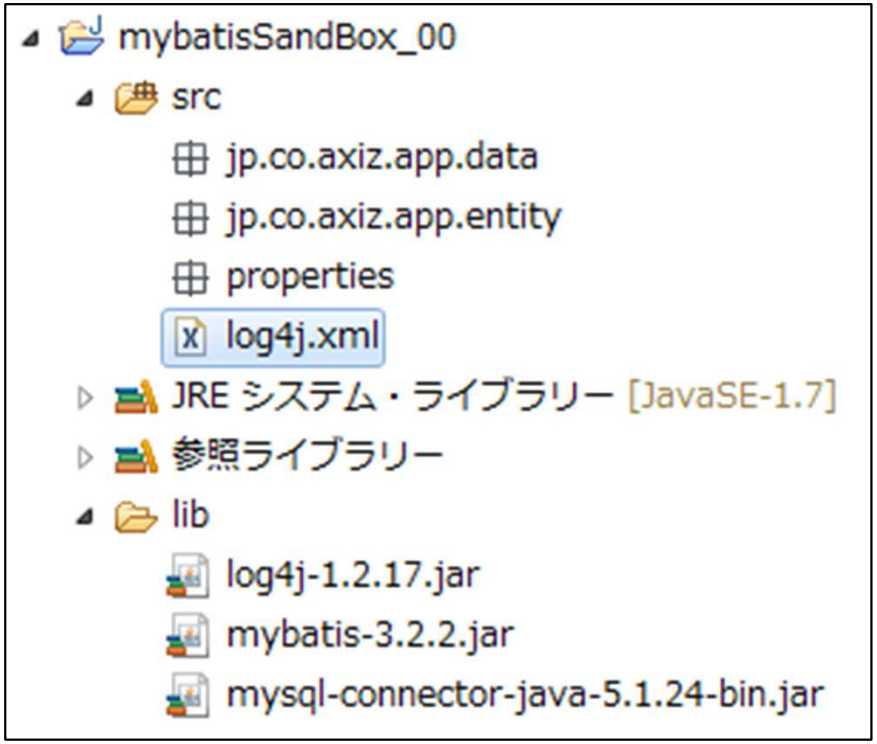
開発環境構築(MyBatis)

開発用プロジェクト作成

右図のパッケージ構成でプロジェクトを作成します。

・libフォルダには、10ページで解凍したライブラリの中から朱書きの.jarファイルを格納してビルド･パスに追加します

・javaソースフォルダ直下には、次ページのlog4j.xmlを配置します



開発環境構築(MyBatis)

【log4j.xml】

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration PUBLIC "-//APACHE//DTD LOG4J 1.2//EN" "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
  <!-- コンソール用アペンダ -->
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d{yyyy-MM-dd HH:mm:ss}-[%5p][%t] %m%n" />
    </layout>
  </appender>

  <!-- Nullアペンダ -->
  <appender name="null" class="org.apache.log4j.varia.NullAppender" />

  <!-- アプリ関連のログ出力 -->
  <logger name="jp.co.axiz">
    <level value="DEBUG" />
    <appender-ref ref="console" />
  </logger>

  <!-- SQL関連のログ出力 -->
  <logger name="java.sql">
    <level value="DEBUG" />
    <appender-ref ref="console" />
  </logger>

  <!-- Rootログ出力 -->
  <root>
    <level value="ERROR" />
    <appender-ref ref="null" />
  </root>
</log4j:configuration>
```

MyBatisによるDBアクセス

データソース・プロパティファイル

前ページまででMyBatisを利用する環境が整いました。
ここからは、この環境を使用して実際にDBアクセス処理を実装していきます。

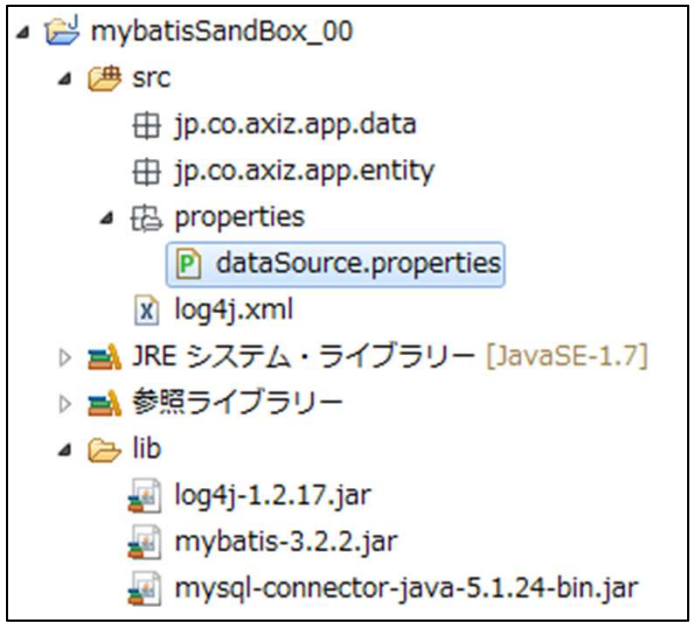
まずは、アクセスするDBサーバの情報をプロパティファイルに記述し、プロジェクトに配置します。

作成するファイルは dataSource.properties(ファイル名は任意)、配置場所はソースフォルダ配下であれば任意ですが、今回は propertiesフォルダとします。

【dataSource.properties】

```
# JDBCドライバ.
driver=com.mysql.jdbc.Driver
# DBサーバ.
url=jdbc:mysql://localhost:3306/axiz_db
# ログインユーザ/パスワード.
username=axizuser
password=axiz
```

☆配置後は以下となります。

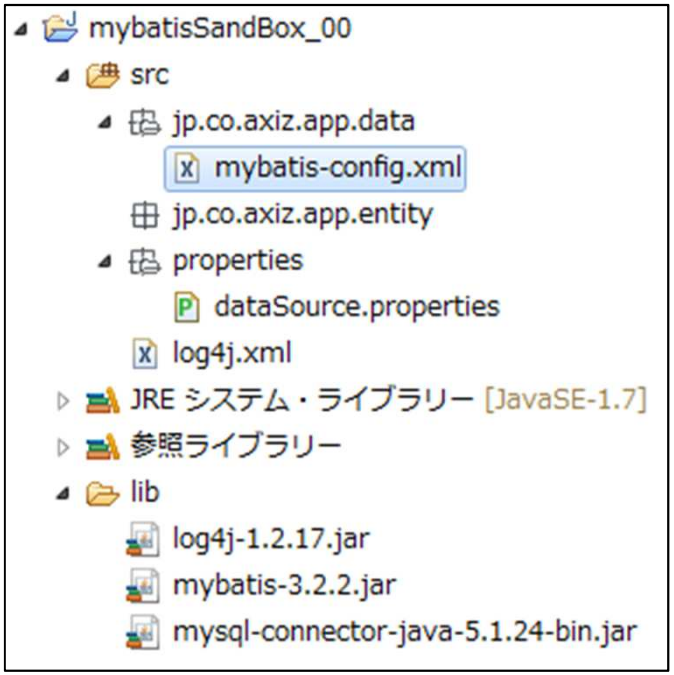


MyBatisによるDBアクセス

MyBatis設定ファイル

次に、MyBatisの設定ファイルを用意します。
作成するファイルは mybatis-config.xml
(ファイル名は任意)、配置は
jp.co.axiz.app.dataパッケージとします。

☆配置後は以下となります。



【mybatis-config.xml】

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
                               "http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

  <!-- データソース・プロパティファイル -->
  <properties resource="properties/dataSource.properties" />

  <!-- DBサーバ接続設定 -->
  <environments default="development">

    <environment id="development">

      <transactionManager type="JDBC" />

      <dataSource type="POOLED">
        <property name="driver"   value="${driver}" />
        <property name="url"      value="${url}" />
        <property name="username" value="${username}" />
        <property name="password" value="${password}" />
      </dataSource>

    </environment>

  </environments>

  <mappers>
  </mappers>

</configuration>
```

dataSource.properties を参照し、
接続先DBサーバを設定しています

MyBatisによるDBアクセス

データ格納用Entityクラス

ここまでで、MyBatisがDBサーバにアクセスするための設定ができました。

では、例としてユーザマスタ(usr_mst)テーブルのレコードを全件抽出する処理を実装していきます。

まずは、ユーザマスタテーブルのレコード(データ)を格納するためのオブジェクト(Entityクラス)を用意します。

作成するファイルは UsrMst.java(クラス名は任意)、配置は jp.co.axiz.app.entityパッケージとします。

なお、フィールド名とカラム名を一致させなければならないということはありません。
クラス名とテーブル名も同様です。

【UsrMst.java】

```
package jp.co.axiz.app.entity;

/**
 * ユーザマスタ({@code usr_mst})テーブルの{@code Entity}クラス。
 * @author {@code AxiZ} t.matsumoto
 */
public class UsrMst {

    /** ユーザ{@code ID}を保持します。 */
    private String usrId;
    /** ログインパスワードを保持します。 */
    private String usrPwd;
    /** 削除済フラグを保持します。 */
    private String isDelFlg;

    public String getUsrId() {
        return usrId;
    }
    public void setUsrId(String usrId) {
        this.usrId = usrId;
    }

    public String getUsrPwd() {
        return usrPwd;
    }
    public void setUsrPwd(String usrPwd) {
        this.usrPwd = usrPwd;
    }

    public String getIsDelFlg() {
        return isDelFlg;
    }
    public void setIsDelFlg(String isDelFlg) {
        this.isDelFlg = isDelFlg;
    }
}
```

MyBatisによるDBアクセス

Mapperインタフェース

テーブルのレコードデータを格納する”入れもの(オブジェクト)”を用意しました。

次に、データ抽出処理を実行する(呼び出す)ためのI/F(インタフェース)を定義します。
今回の例であれば「ユーザマスタ情報全件検索メソッド」のI/Fを用意することになります。

「利用側はこのメソッドを呼び出せば、”ユーザマスタ情報一覧”を取得することができる」というシグネチャで定義します。

作成するファイルは UsrMstMapper.java(インタフェース名は任意)、配置はjp.co.axiz.app.dataパッケージとします。

【UsrMstMapper.java】

```
package jp.co.axiz.app.data;

import java.util.List;

import jp.co.axiz.app.entity.UsrMst;

/**
 * ユーザマスタ({@code usr_mst})テーブルの{@code Mapper}インタフェース。
 *
 * @author {@code AxiZ} t.matsumoto
 */
public interface UsrMstMapper {

    /**
     * ユーザマスタのレコードを全件取得します。<p />
     *
     * 返却する一覧は、ユーザ{@code ID}(昇順)でソートされています。<br />
     * レコードが存在しない場合は、要素{@code 0}の{@link List}を返却します。
     *
     * @return ユーザマスタ情報一覧
     */
    List<UsrMst> findAll();
}
```

MyBatisによるDBアクセス

マッピングファイル

Mapper I/Fを定義することで、テーブルデータアクセスのクチを開けました。

いよいよ、MyBatisの核心部分であるマッピングファイルを作成します。

このマッピングファイルには、以下の定義を記述します。

・抽出データとEntityとのマッピング
・実行するSQL
・SQLとMapper I/Fとのマッピング

作成するファイルは UsrMstMapper.xml、配置はjp.co.axiz.app.dataパッケージとします。

ファイル名は任意ですが、メンテナンス性を考慮し、対応するMapper I/Fと同名が良いでしょう。

また、マッピングファイルとMapper I/Fのセットは基本的に1テーブルに1セット用意します。

【UsrMstMapper.xml】

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
                        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- ユーザマスタ(usr_mst)テーブルMapper定義 -->
<mapper namespace="jp.co.axiz.app.data.UsrMstMapper">

  <!-- ================================================ ResultMaps -->

  <!-- ユーザマスタEntity -->
  <resultMap id="usrMstResultMap" type="jp.co.axiz.app.entity.UsrMst">
    <id     column="usr_id"     property="usrId" />
    <result column="usr_pwd"    property="usrPwd" />
    <result column="is_del_flg" property="isDelFlg" />
  </resultMap>

  <!-- ================================================ Queries -->

  <!-- ユーザマスタ情報全件取得 -->
  <select id="findAll" resultMap="usrMstResultMap">
    SELECT usr_id,
           usr_pwd,
           is_del_flg
      FROM usr_mst
     ORDER BY usr_id
  </select>

</mapper>
```

* 今回、MyBatis と Spring Frameworkの連携を行います
* そのため、(まずは)便宜上Mapper I/Fとマッピングファイルは同名とし、格納場所も同ディレクトリとします

MyBatisによるDBアクセス

マッピングの適用

データアクセスのためのリソースの作成と、マッピングが完了しました。

最後に、MyBatis設定ファイル内にマッピングファイルを定義してマッピングを適用させます。

編集するファイルは mybatis-config.xml です。

<mappers></mappers>の中に、作成した UsrMstMapper.xml を定義します。

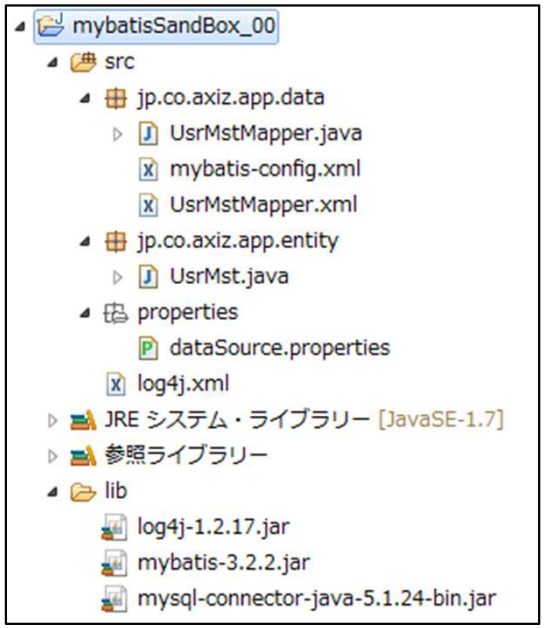
なお、マッピングファイルの定義を追加する場合は、mappers エレメント内に追加していきます。

```
<mappers>
  <mapper resource="xxxxx" />
  <mapper resource="yyyyy" />
</mappers>
```

【mybatis-config.xml】

```
<configuration>

  <properties resource=..... />

  <environments default=.....>
    .....
  </environments>

  <mappers>
    <!-- ユーザマスタ(usr_mst)TBLマッピングファイル -->
    <mapper resource="jp/co/axiz/app/data/UsrMstMapper.xml" />
  </mappers>

</configuration>
```

☆パッケージ内は以下の構成となります。



MyBatisによるDBアクセス

DBアクセスの実行

以上で全ての準備が整いました。

では、ユーザマスタ全件検索メソッドを呼び出して、ユーザマスター一覧を取得します。

動作確認用にmainメソッドからUsrMstMapper#findAll()を呼び出してみましょう（ソースは次ページ）。

作成するファイルは UsrMstTest.java、配置はjp.co.axiz.appパッケージとします(スペースの都合上import宣言は一部省略していますので、適宜挿入してください)。

60行目辺りのTODOコメント部分に、Entityの各フィールド値を(標準出力に)出力するコードを実装後、実行してみてください。

Eclipseのコンソール･ビュー上にクエリ実行ログと、取得したユーザマスタ情報一覧が出力されていれば成功です。

MyBatisによるDBアクセス

DBアクセスの実行

【UsrMstTest.java】

```
package jp.co.axiz.app;

/**
 * ユーザマスタ({@code usr_mst})テーブルアクセスの動作確認テストクラス。
 * @author {@code AxiZ} t.matsumoto
 */
public class UsrMstTest {
    /** {@code mybatis-config.xml}のファイルパスを表す定数。 */
    private static final String MYBATIS_CONFIG = "jp/co/axiz/app/data/mybatis-config.xml";

    /** {@link SqlSessionFactory}オブジェクトを保持します。 */
    private static SqlSessionFactory sessionFactory;

    /**
     * エントリ･ポイントとしての処理を実行します。
     * @param args デフォルト･パラメータ
     */
    public static void main(String[] args) {
        Reader reader;
        try {
            reader = Resources.getResourceAsReader(MYBATIS_CONFIG);
        } catch (IOException e) {
            e.printStackTrace();
            return;
        }
        sessionFactory = new SqlSessionFactoryBuilder().build(reader);
        getUsrMstAll();
    }

    /**
     * ユーザマスタのレコードを全件取得します。
     * @see UsrMstMapper#findAll()
     */
    private static void getUsrMstAll() {
        final SqlSession sqlSession = sessionFactory.openSession();
        final List<UsrMst> resultList;
        try {
            resultList = sqlSession.getMapper(UsrMstMapper.class).findAll();
        } finally {
            sqlSession.close();
        }
        for (final UsrMst entity : resultList) {
            // TODO: ユーザマスタ情報出力処理を実装すること.
        }
    }
}
```

MyBatisによるDBアクセス

SQLパラメータバインド

データ操作には、検索条件や更新データ等、パラメータが動的に変化する場合があります。
今回は「指定したユーザIDに合致するユーザマスタレコード件数取得」を例として、パラメータバインドを説明していきます。

まず、Mapper I/Fに下記のメソッドを定義してください。
次に、マッピングファイルに、実行するSQLを定義します。

【UsrMstMapper.java】

```
/**
 * 指定したユーザ{@code ID}に合致するレコード数を取得します。
 *
 * @param usrId ユーザ{@code ID}
 * @return レコード数
 */
int getCountById(String usrId);
```

【UsrMstMapper.xml】

```
<!-- レコード数取得(ユーザID指定) -->
<select id="getCountById" parameterType="string" resultType="int">
  SELECT COUNT(usr_id)
    FROM usr_mst
   WHERE usr_id = #{usrId}
</select>
```

#getCountById(String) でString型の引数を受け取るため、parameterType属性には"string"または"java.lang.String" を指定し、このパラメータがSQL内の #{usrId} にバインドされます。

また、このSQLの実行結果は件数なので、resultType属性には "int" を指定しています。

では、UserMstMapper#getCountById(String)を呼び出してみましょう。

正しい抽出件数が取得できれば成功です。
クエリ実行ログでも、パラメータ値と抽出件数が確認できます。

MyBatisによるDBアクセス

動的SQLの構築 1/4

動的パラメータによるSQLバインドは実装できました。

しかしパラメータだけではなく、条件によってSQLそのものを動的に構築する場合もあります。

次は「指定した検索条件に合致するユーザ詳細情報一覧取得」を例として、動的SQL構築を説明していきます。

まず、ユーザ詳細情報格納用の Entityを用意します。

【UsrDetail.java】

```
package jp.co.axiz.app.entity;

/**
 * ユーザ詳細情報{@code Entity}クラス。
 *
 * @author {@code AxiZ} t.matsumoto
 */
public class UsrDetail {

    /** ユーザ{@code ID}を保持します。 */
    private String usrId;
    /** ログインパスワードを保持します。 */
    private String usrPwd;
    /** ユーザ名を保持します。 */
    private String usrName;
    /** 電話番号を保持します。 */
    private String usrTel;
    /** 住所を保持します。 */
    private String usrAddr;
    /** 削除済フラグを保持します。 */
    private String isDelFlg;

    // TODO: 各アクセサを実装すること.
}
```

今回の処理では、ユーザマスタ(usr_mst)テーブルとユーザ情報(usr_inf)テーブルを結合して抽出したデータを”ユーザ詳細情報”とします。

そのため、プロパティは両テーブルのカラムをマージしたものとなっています。

* アクセサは省略しているので、適宜実装してください

MyBatisによるDBアクセス

動的SQLの構築 2/4

次に、Mapper I/Fに下記のメソッドを定義してください。

【UsrMstMapper.java】

```
/**
 * 指定された条件に合致するユーザ詳細情報一覧を取得します。<p />
 *
 * <ul>
 *   <li>返却する一覧は、以下の条件でソートされています。
 *     <table>
 *       <tr>
 *         <td>{@code 1.}</td><td>ユーザ{@code ID}</td><td>(昇順)</td>
 *       </tr>
 *       <tr>
 *         <td>{@code 2.}</td><td>ユーザ名</td><td>(昇順)</td>
 *       </tr>
 *     </table>
 *   </li>
 *   <li>条件に合致するユーザ詳細情報が存在しない場合は、要素{@code 0}の{@link List}を返却します。</li>
 * </ul>
 *
 * @param param 検索条件
 * @return ユーザ詳細情報一覧
 */
List<UsrDetailEntity> findDetailListByConditions(UsrDetailEntity param);
```

検索条件を格納した UsrDetail を引数として受け取ります。

設計上、別途パラメータ専用のDTOを定義して、そちらを使用しても構いません。

今回は UsrMstMapper内に追加定義していますが、設計としてユーザ詳細情報用に別途 Mapper I/Fを作成しても、もちろん構いません。

MyBatisによるDBアクセス

動的SQLの構築 3/4

最後に、マッピングファイルに Entity と SQL を定義します。

今回の「ユーザ詳細情報一覧取得」では、対象レコードを "未削除ユーザ(is_del_flg = '0')" に限定しています。

また、検索条件は

・ユーザID(前方一致)
・ユーザ名(部分一致)
・電話番号(部分一致)
・住所(部分一致)

としています。

このSQLでは、WHERE節を動的に構築しています。 具体的には、各条件ごとに値が存在する場合のみWHERE節に追加し、値をバインドしています。

JSTLタグによく似た記述形式です。

【UsrMstMapper.xml】

```
<!-- ユーザ詳細情報Entity -->
<resultMap id="usrDetailResultMap" type="jp.co.axiz.app.entity.UsrDetail">
  <id     column="usr_id"     property="usrId" />
  <result column="usr_pwd"    property="usrPwd" />
  <result column="usr_name"   property="usrName" />
  <result column="usr_tel"    property="usrTel" />
  <result column="usr_addr"   property="usrAddr" />
  <result column="is_del_flg" property="isDelFlg" />
</resultMap>

<!-- ユーザ詳細情報取得(一覧) -->
<select id="findDetailListByConditions" resultMap="usrDetailResultMap"
        parameterType="jp.co.axiz.app.entity.UsrDetail">
  SELECT mst.usr_id     AS usr_id,
         mst.usr_pwd    AS usr_pwd,
         inf.usr_name   AS usr_name,
         inf.usr_tel    AS usr_tel,
         inf.usr_addr   AS usr_addr,
         mst.is_del_flg AS is_del_flg
    FROM usr_mst mst
    JOIN usr_inf inf ON mst.usr_id=inf.usr_id
   WHERE mst.is_del_flg='0'
  <if test="usrId != null">
     AND mst.usr_id     LIKE CONCAT(#{usrId}, '%')
  </if>
  <if test="usrName != null">
     AND inf.usr_name   LIKE CONCAT('%', #{usrName}, '%')
  </if>
  <if test="usrTel != null">
     AND inf.usr_tel    LIKE CONCAT('%', #{usrTel}, '%')
  </if>
  <if test="usrAddr != null">
     AND inf.usr_addr   LIKE CONCAT('%', #{usrAddr}, '%')
  </if>
   ORDER BY usr_id, usr_name
</select>
```

MyBatisによるDBアクセス

動的SQLの構築 4/4

先ほどのケースでは、固定のWHERE節が存在しましたが、条件によってWHERE節の有無が動的に変わる場合もあります。

その場合は、以下のように<where>タグを使用することで対応できます。

```
<select ...>
  SELECT ...
    FROM ...
  <where>
    <if test="xxx != null">AND XXX = #{xxx}</if>
    <if test="yyy != null">AND YYY = #{yyy}</if>
    <if test="zzz != null">AND ZZZ = #{zzz}</if>
  </where>
</select>
```

<where>タグは、要素内の<if>タグが結果を返す場合のみWHERE を挿入します。

また、返された結果が AND または OR で始まっている場合は、これを除去します。

同じような働きをするタグに UPDATE文に使用する<set>タグがあります。

```
<upadte ...>
  UPDATE ...
  <set>
    <if test="xxx != null">xxx = #{xxx},</if>
    <if test="yyy != null">yyy = #{yyy},</if>
    <if test="zzz != null">zzz = #{zzz},</if>
  </set>
  WHERE ...
</select>
```

<set>タグは、要素内の<if>タグが結果を返す場合のみSET を挿入します。

また、SET節末尾の ,(カンマ)を除去します。