データベースからデータを取得するビジネスロジックの実装例を次に示します。

```
 1: public class ShipmentResultListSearchLogic extends BaseQueryLogic
 2: {
 3:   public void logicSearch(ShipmentEntryListForm form,ActionErrors errors,ShipmentResultListView objView)throws XSDBException
 4:   {
 5:     // エラーが発生している場合、結果出力をしない・・・①
 6:     if ( !errors.isEmpty() ) {
 7:       objView.setShipmentEntryListData(null);
 8:       return;
 9:     }
10:
11:     // Sessionよりログイン情報データオブジェクトを取得・・・②
12:     LoginInfoView loginInfo = (LoginInfoView)request.getObject(RequestUtil.SCOPE_SESSION, LoginInfoView.class );
13:
14:     // SQL作成・・・③
15:     bindUtil.append("select ");
16:     bindUtil.append(" A.ITEM_CD, A.ITEM_NM, A.SIIRE_CD, A.SIIRE_NM, A.ITEM_JAN_CD ");
17:     bindUtil.append(",A.STOCK_NUM AS SHIPMENT_NUM ");//出庫数
18:     bindUtil.append(",(NVL(B.STOCK_NUM,0) + NVL(A.STOCK_NUM,0)) AS STOCK_NUM ");//出庫前在庫数
19:     bindUtil.append(",B.STOCK_NUM AS TOTAL_STOCK_NUM ");//現在在庫数
20:     bindUtil.append("from TR_STOCK_HISTORY A, TR_STOCK B ");
21:     bindUtil.append("where A.CLIENT_CD = B.CLIENT_CD ");
22:     bindUtil.append("and A.SALON_CD = B.SALON_CD ");
23:     bindUtil.append("and A.ITEM_CD = B.ITEM_CD ");
24:     bindUtil.append("and A.CLIENT_CD=? ", loginInfo.getCLIENT_CD());
25:     bindUtil.append("and A.SALON_CD=? ", loginInfo.getSALON_CD());
26:     bindUtil.append("and A.HISTORY_KBN=? ", LiteralConstants.STOCK_HISTORY_KBN_SHIPMENT);
27:     bindUtil.append(bindUtil.repeatIN(" and ", "A.ITEM_CD",form.getItemCd()));
28:     bindUtil.append(bindUtil.repeatIN(" and ", "A.STOCK_SEQ",form.getItemSeqNo()));
29:     bindUtil.append("ORDER BY A.STOCK_SEQ");
30:
31:     // 出庫商品リストデータのインスタンス生成・・・④
32:     ShipmentEntryListData shipmentEntryListData = new ShipmentEntryListData();
33:
34:     // 問い合わせ実行・・・⑤
35:     Collection collection = query(shipmentEntryListData);
36:     shipmentEntryListData.inCollection(collection);
37:
38:     // 出庫商品リストをViewにセット・・・⑥
39:     objView.setShipmentEntryListData(shipmentEntryListData);
40:     // 合計行をViewにセット・・・⑦
41:     objView.setSUM_SHIPMENT_NUM(CollectionUtil.getSummaryFormat(collection, "SHIPMENT_NUM"));//数量
42:     objView.setSUM_STOCK_NUM(CollectionUtil.getSummaryFormat(collection, "STOCK_NUM"));//出庫前在庫数
43:     objView.setSUM_TOTAL_STOCK_NUM(CollectionUtil.getSummaryFormat(collection, "TOTAL_STOCK_NUM"));//出庫後在庫数
44:   }
45: }
```

① このロジックが呼び出される前にエラーが発生していた場合、引数errorsにはエラーが格納されています。
このロジック例では、既にエラーが発生していた場合、ビューのShipmentEntryListDataプロパティにnullを設定してロジックを終了しています。

② requestは、基底クラスBaseQueryLogicで定義されている、RequestUtil型のprotectedフィールドです。
RequestUtilは、HttpServletRequestのラッパークラスで、リクエストやセッションへのアクセスを簡略化します。
12行目の例では、LoginInfoViewクラスの完全修飾クラス名"login.LoginInfoView"をキーにして、セッションからオブジェクトを取得しています。

スコープには次の3種類が指定できます。
・SCOPE_REQUEST・・・リクエスト
・SCOPE_SESSION・・・セッション
・SCOPE_CONTEXT・・・コンテキスト

また、setObjectメソッドで、オブジェクトを任意のスコープに格納することもできます。

③ bindUtilは、基底クラスBaseQueryManager（BaseQueryLogicの基底クラス）で定義されている、PreparedBindUtil型のprotectedフィールドです。
PreparedBindUtilクラスは、SQL文の構築をサポートするクラスです。

＜SQL文字列の連結 - append＞
appendメソッドでSQL文字列を連結します。24行目の例のように、パラメータをバインドすることもできます。

bindUtil.append("and A.CLIENT_CD=? ", loginInfo.getCLIENT_CD());

loginInfo.getCLIENT_CD()が示す値が、SQL文字列中の"?"にバインドされます。
1回のappendメソッド呼び出しで、最大3つまでパラメータをバインドできます。

appendメソッドで連結した際、文字列の末尾に改行コードが付加されます。したがって、

```
bindUtil.append("SELECT A");
bindUtil.append("FROM B");
```

とした場合でも、"SELECT AFROM B"とはなりません。

＜IN条件文の生成 - repeatIN＞
WHERE句のIN条件を記述する場合、不特定多数のパラメータをバインドする必要があります。
このような場合には、27行目の例のように、repeatINメソッドを使用します。

```
bindUtil.append(bindUtil.repeatIN(" and ", "A.ITEM_CD",form.getItemCd()));
```

form.getItemCd()はString型の配列を表します。たとえばこの配列が要素を3つ含む場合、bindUtil.repeatINは次の文字列を返します。

```
" and A.ITEM_CD IN(?,?,?)"
```

また、この配列の3つの要素がSQL文にバインドされます。

詳細はPreparedBindUtilクラスのjavadocドキュメントを参照してください。

④ データベースの検索結果を格納するクラスのインスタンスを作成します。

⑤ データベース検索を行います。

```
Collection collection = query(shipmentEntryListData);
```

queryメソッドは、基底クラスBaseQueryManager（BaseQueryLogicの基底クラス）で定義されています。
queryメソッドはbindUtilで作成したSQL文を実行します。
取得したデータは、引数で指定されたオブジェクトの型を要素型とするコレクションとして返されます。（引数で指定されたオブジェクトには取得結果は格納されません。）
上記の例では、ShipmentEntryListDataクラスのインスタンスが、取得したレコードの件数分生成され、コレクションに格納されて返されます。
取得したレコードは、ShipmentEntryListDataクラスのインスタンスにマッピングされます。

36行目では、取得したコレクションをshipmentEntryListDataにセットしています。これにより、検索結果がshipmentEntryListDataに格納されます。

```
shipmentEntryListData.inCollection(collection);
```

⑥ 検索結果を格納したデータオブジェクトを、ビューに設定します。

画面に表示するためのデータはビューに格納します。ただし、入力フォームにデフォルト値を設定するような場合には、フォームに格納することもあります。

⑦ 検索結果から値を計算し、ビューのプロパティに設定します。

```
CollectionUtil.getSummaryFormat(collection, "SHIPMENT_NUM")
```

CollectionUtilクラスは、コレクションを操作するユーティリティクラスです。
getSummaryFormatメソッドは、コレクション要素の任意のプロパティを合計した値を返します。
第1引数には、集計対象のコレクションを指定します。この例ではShipmentEntryListData型のオブジェクトを要素とするコレクションを対象としています。
上記の例では、コレクションの各要素のSHIPMENT_NUMプロパティを集計し、合計値を返します。

詳細はCollectionUtilクラスのjavadocドキュメントを参照してください。

データベースを更新するビジネスロジックの実装例を次に示します。

```
1: public class StockModifyEntryLogic extends BaseTranLogic
2: {
3:   public void logicEntry(StockSearchForm form,ActionErrors errors)throws XSDBException,XSDBDuplicateException
4:   {
5:     // Sessionよりログイン情報データオブジェクトを取得・・・①
6:     LoginInfoView loginInfo = (LoginInfoView)request.getObject(RequestUtil.SCOPE_SESSION,LoginInfoView.class);
7:
8:     try
9:     {
10:      // ResultSetMap
11:      ResultSetMap rsm = null;
12:
13:      /* 排他制御処理 */
14:      // SQL生成
15:      bindUtil.append("SELECT ");
16:      bindUtil.append("  CLIENT_CD,SALON_CD,ITEM_CD");
17:      bindUtil.append("FROM ");
18:      bindUtil.append("  TR_STOCK ");
19:      bindUtil.append("WHERE ");
20:      bindUtil.append("  CLIENT_CD =? ",loginInfo.getCLIENT_CD());
21:      bindUtil.append("AND SALON_CD =? ",form.getCorrectSalonCd());
22:      bindUtil.append("AND ITEM_CD =? ",form.getCorrectItemCd());
23:      rsm = queryLock(false);・・・②
24:
25:      /* 在庫訂正対象データ検索処理 */
26:      // 在庫訂正データ検索SQL生成
27:      bindUtil.append("SELECT ");
28:      bindUtil.append("A.ITEM_CD,B.ITEM_NM,B.ITEM_BG_CD,A.SIIRE_CD,");
29:      bindUtil.append("A.SIIRE_NM,B.ITEM_JAN_CD,A.STOCK_NUM,B.URI_TAN ");
30:      bindUtil.append("FROM ");
31:      bindUtil.append("TR_STOCK A,MT_ITEM B ");
32:      bindUtil.append("WHERE ");
33:      bindUtil.append("A.CLIENT_CD =? ",loginInfo.getCLIENT_CD());
34:      bindUtil.append("AND A.SALON_CD =? ",form.getCorrectSalonCd());
35:      bindUtil.append("AND A.ITEM_CD =? ",form.getCorrectItemCd());
36:      bindUtil.append("AND A.ITEM_CD=B.ITEM_CD");
37:      // 問い合わせ実行
38:      rsm = query();・・・③
39:
40:      /* 在庫データ更新処理 */
41:      // 在庫データ更新SQL生成
42:      bindUtil.append("UPDATE TR_STOCK ");
43:      bindUtil.append("SET ");
44:      bindUtil.append("STOCK_NUM=?,",form.getCorrectStockNum());
45:      bindUtil.append("UP_DT=?,",new Date());
46:      bindUtil.append("UP_USER_ID=? ",loginInfo.getUSER_ID());
47:      bindUtil.append("WHERE ");
48:      bindUtil.append("CLIENT_CD=? ",loginInfo.getCLIENT_CD());
49:      bindUtil.append("AND SALON_CD=? ",form.getCorrectSalonCd());
50:      bindUtil.append("AND ITEM_CD=?",form.getCorrectItemCd());
51:      // 在庫データ更新処理実行
52:      update();・・・④
53:
54:      /* 在庫変更履歴登録処理 */
55:      rsm.next();
56:      // 在庫変更履歴登録SQL生成
57:      bindUtil.append("INSERT INTO ");
58:      bindUtil.append("TR_STOCK_HISTORY(");
59:        :
60:        :
61:      bindUtil.append("(to_char(?,'YYYYMM')",new Date());
62:      bindUtil.append("||LPAD(HISTORY_SEQ.nextval,?,",LiteralConstants.REPLACE_LENGTH);
63:      bindUtil.append("?),",LiteralConstants.REPLACE_ZERO);
64:      bindUtil.append("?,",LiteralConstants.STOCK_HISTORY_KBN_MODIFY);
65:        :
66:        :
67:      bindUtil.append(")");
68:      // 在庫変更履歴処理実行
69:      insert();・・・⑤
70:    }
71:    catch (XSDBNotFoundException e)・・・⑥
72:    {
73:      errors.add("",new ActionError("validateError.app.notFoundData"));
74:    }
75:    catch (XSDBLockException e)
```

```
76:     {
77:       errors.add("", new ActionError("db.lock.error"));
78:     }
79:   }
80: }
```

① 参照系ビジネスロジックの説明を参照してください。

② レコードをロックします。
queryLockメソッドは、基底クラスBaseQueryManager（BaseQueryLogicの基底クラス）で定義されています。
引数には、すでにロックされていた場合のウェイトを指定します。
・true・・・ロックが解除されるまで待ちます。
・false・・・即座にXSDBLockExceptionをスローします。

上記の例では、すでにロックされていた場合にXSDBLockExceptionをスローします。

③ 参照系ビジネスロジックの説明を参照してください。
ここで再度bindUtilを使用していますが、queryLockメソッドを呼び出した時点で、bindUtilのSQL文字列バッファはクリアされていますので、新たに作成されたSQL文字列に対してqueryメソッドが実行されます。また、queryメソッド実行時にもbindUtilのバッファがクリアされます。

④ データを更新します。
updateメソッドは、基底クラスBaseQueryManager（BaseQueryLogicの基底クラス）で定義されています。
updateメソッドはqueryメソッドと同様に、bindUtilで作成したSQL文（update文）を実行します。

⑤ データを追加します。
insertメソッドは、基底クラスBaseQueryManager（BaseQueryLogicの基底クラス）で定義されています。
insertメソッドもqueryメソッドと同様に、bindUtilで作成したSQL文（insert文）を実行します。

⑥ データベースアクセス時に発生する可能性のある例外を捕捉します。
データベース例外をアプリケーション例外に変換します。新しく生成したActionErrorオブジェクトを、ロジックメソッドの引数errorsに登録します。ActionErrorはStrutsで提供されているクラスです。

データベースアクセス時の主な例外を以下に挙げます。

・XSDBException・・・データベースの基底例外
・XSDBDuplicateException・・・一意制約違反
・XSDBLockException・・・ロック競合
・XSDBNotFoundException・・・データなし
・XSDBParameterException・・・パラメータ不正

データベースを更新するビジネスロジックの別の実装例を次に示します。

```
1: public class OrderAnalysisEntryLogic extends BaseTranLogic
2: {
3:   public void logicEntry(OrderAnalysisSearchForm form, ActionErrors errors)
4:       throws XSDBException,XSDBLockException,XSDBDuplicateException,XSDBParameterException
5:   {
6:     // Sessionよりログイン情報データオブジェクトを取得
7:     LoginInfoView loginInfo = (LoginInfoView)request.getObject(RequestUtil.SCOPE_SESSION,LoginInfoView.class);
8:     // 更新用レコード格納クラスのインスタンス作成
9:     OrderAnalysisListDataRec listDataRec = new OrderAnalysisListDataRec();
10:
11:    // 更新データの作成・・・①
12:    String queryValue[] = form.getQueryValue();
13:    String clientCd = loginInfo.getCLIENT_CD();
14:    String salonCd[] = new String[queryValue.length];
15:    String itemCd[] = new String[queryValue.length];
16:    for (int i=0; i<form.getQueryValue().length; i++) {
17:      //データレコードのインスタンス作成
18:      OrderAnalysisListDataRec dataRec = new OrderAnalysisListDataRec();
19:      //キー項目の取得
20:      String value[] = StringUtil.split(queryValue[i], LiteralConstants.JOIN_SEPARATOR);
21:      salonCd[i] = value[LiteralConstants.ANARYSIS_INDEX_SALON_CD];
22:      itemCd[i] = value[LiteralConstants.ANARYSIS_INDEX_ITEM_CD];
23:      //データレコードを設定
24:      dataRec._setCLIENT_CD(clientCd);
25:      dataRec._setSALON_CD(salonCd[i]);
26:      dataRec._setITEM_CD(itemCd[i]);
27:      dataRec.setORDER_LIMIT_NUM(Integer.parseInt(form.getOrderLimitNum()[i]));
28:      dataRec.setORDER_NUM(Integer.parseInt(form.getDefaultOrderNum()[i]));
29:      dataRec.setUP_DT(DateUtil.getDayTime());
30:      dataRec.setUP_USER_ID(loginInfo.getUSER_ID());
31:      //データレコードをコレクションに格納
32:      listDataRec.append(dataRec);
33:    }
34:
35:    // 更新処理実行
36:    try {
37:      //更新対象レコードのロック・・・②
38:      bindUtil.append("select *");
39:      bindUtil.append("  from TR_STOCK");
40:      bindUtil.append(" where CLIENT_CD=?", clientCd);
41:      bindUtil.append(bindUtil.repeatIN(" and ", "SALON_CD", salonCd));
42:      queryLock(false);
43:
44:      //更新処理実行・・・③
45:      Collection dataRecCollection = listDataRec.outCollection();
46:      update(dataRecCollection);
47:    }
48:    catch (XSDBLockException e) {
49:      errors.add("",new ActionError("db.error.lock"));
50:    }
51:    catch (XSDBNotFoundException e) {
52:      errors.add("",new ActionError("validateError.app.notFoundData"));
53:    }
54:    catch (XSDBDuplicateException e) {
55:      errors.add("",new ActionError("db.error.duplicate"));
56:    }
57:    catch (XSDBParameterException e) {
58:      errors.add("",new ActionError("db.error.duplicate"));
59:    }
60:    catch (XSDBException e) {
61:      errors.add("",new ActionError("error.parameter.exception"));
62:    }
63:  }
64: }
```

① 更新データを作成します。ここで作成したデータは、③のデータ更新の際に使用します。
上記の例では、OrderAnalysisListDataRec型のオブジェクトをレコード件数分作成しています。

② レコードをロックします。

③ データを更新します。Collection型を引数とするupdateメソッドを実行しています。
このupdateメソッドも、基底クラスBaseQueryManager（BaseQueryLogicの基底クラス）で定義されています。
コレクションの要素は以下の規則に従って作成する必要があります。

・更新するテーブル名を返す、"_defTableName"という名称のメソッドを定義します。
・主キー列を示す"_set[列名]"、"_get[列名]"という名称のメソッドを定義します。
・更新対象の列を示す"set[列名]"、"get[列名]"という名称のメソッドを定義します。

updateメソッドを実行すると、主キーを元にレコード検索し、更新対象の列を更新するSQL文を発行します。

上記以外にも動作に細かい調整を行うことができます。詳細はBaseQueryManagerのjavadocドキュメントを参照してください。

<options>

Collectionオブジェクトから HTMLの<option>タグを生成します。
（Strutsの<html:options>タグの機能を拡張したものです。）

属性	必須	説明
name		Collectionオブジェクトを格納しているBean名を指定します。
property	○	Beanのプロパティ名。プロパティはCollection型のプロパティを指定します。
labelProperty		画面に表示する項目名を示すプロパティの名前を指定します。
valueProperty		<option>タグのvalue属性に設定する値を示すプロパティの名前を指定します。

<command>

Mediatorの動作遷移のキーとなるリクエストパラメータ（command、mode）を生成します。

属性	必須	説明
name		commandパラメータを指定します。
mode		modeパラメータを指定します。

<propertyList>

プロパティファイルからHTMLの<option>タグを生成します。

属性	必須	説明
key	○	プロパティのキーを指定します。
propertyFile	○	プロパティファイル名を指定します。
startPosition		<option>タグの生成を開始するプロパティのインデックスを指定します。
endPosition		<option>タグの生成を終了するプロパティのインデックスを指定します。
labelValue		画面に表示する項目名をvalue属性に設定するかどうかを指定します。

例）

list.propertiesファイル

```
item.select.period.0=3ヶ月
item.select.period.1=6ヶ月
item.select.period.2=1年
```

JSP

```
<html:select property="list">
  <taglib:propertyList key="item.select.period" propertyFile="list.properties"
     startPosition="0" endPosition="2" labelValue="true" />
</html:select>
```

生成されるHTML

```
<select name="list">
  <option value="3ヶ月">3ヶ月</option>
  <option value="6ヶ月">6ヶ月</option>
  <option value="1年">1年</option>
</select>
```

<write>

Beanのプロパティを出力します。
（Strutsの<bean:write>タグの機能を拡張したものです。）

属性	必須	説明
filter		trueの場合、出力する値をフィルタ処理し、HTMLの予約語をエンコードします。

ignore		trueの場合、Beanがスコープに見つからないときに何も出力しません。falseの場合、Beanがスコープに見つからないときに例外をスローします。
name	○	Beanの名称を指定します。
property		Beanのプロパティ名を指定します。
scope		Beanのスコープを指定します。
escape		trueの場合、Beanの値がnullのときに" "を出力します。falseの場合、Beanの値がnullのときには何も出力しません。
prefix		プロパティにアクセスするメソッドの接頭辞を指定します。デフォルトは"get"で、"getXXX"メソッドでプロパティにアクセスします。