

**TERASOLUNA Server Framework for Java
(Web 版)**

機能説明書

第 2.0.1.0 版

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、「参考文献:TERASOLUNA Server Framework for Java (Web 版) 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。

- Apache、Tomcat は、Apache Software Foundation の登録商標または商標です。
- Java、JDK、J2SE、J2EE、JSP、Servlet は、米国 Sun Microsystems, Inc.の米国及びその他の国における登録商標または商標です。
- Oracle は、米国 Oracle International Corp.の米国及びその他の国における登録商標または商標です。
- TERASOLUNA は、株式会社 NTT データの登録商標です。
- WebLogic は、BEA Systems Inc.の登録商標または商標です。
- Windows は、米国 Microsoft Corp.の米国及びその他の国における登録商標または商標です。
- Cosminexus は、株式会社日立製作所の登録商標です。
- 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。

本書は、TERASOLUNA Server Framework for Java (Web 版) ver2.0.1.0 に対応しています。

目次

● 共通機能

CA-01	トランザクション管理機能
CB-01	データベースアクセス機能
CC-01	JNDI アクセス機能
CD-01	ユーティリティ機能

● Web 版機能

WA-01	ログオン済みチェック機能
WA-02	アクセス権限チェック機能
WA-03	サーバ閉塞チェック機能
WA-04	業務閉塞チェック機能
WA-05	拡張子直接アクセス禁止機能
WB-01	ユーザ情報保持機能
WB-02	アクションフォーム拡張機能
WB-03	アクションフォーム切替機能
WB-04	フォームプロパティリセット機能
WB-05	コードリスト機能
WC-01	Action 実行時システム例外処理機能
WD-01	セッションディレクトリ機能
WD-02	ファイルアップロード機能
WD-03	ファイルダウンロード機能
WE-01	アクション拡張機能
WE-02	標準ディスプレイパッチャ機能
WE-03	フォワード機能
WE-04	コードリスト再読み込み機能
WE-05	セッションディレクトリ作成機能
WE-06	セッションクリア機能
WE-07	ログオフ機能
WF-01	拡張入力チェック機能
WG-01	拡張メッセージリソース機能
WH-01	ビジネスロジック実行機能
WH-02	ビジネスロジック入出力機能
WI-01	一覧表示機能
WJ-01～WK-09	画面表示機能

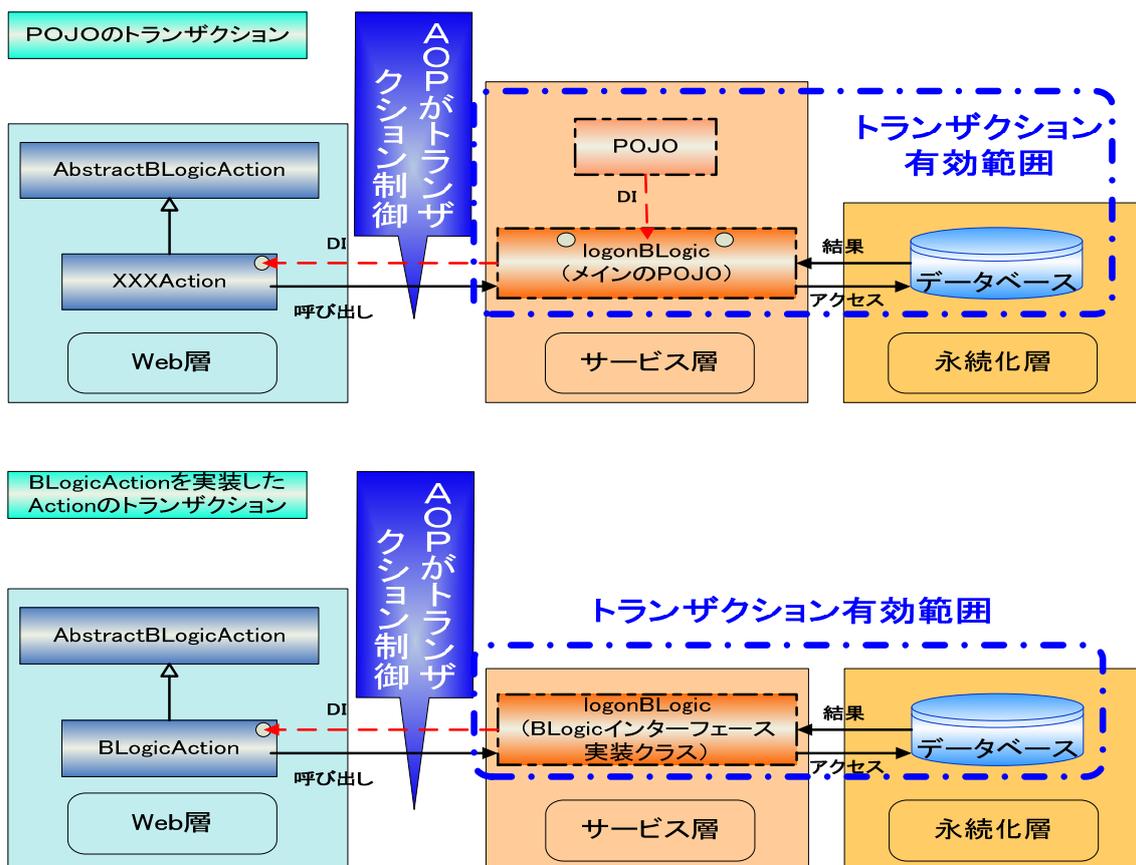
CA-01 トランザクション管理機能

■ 概要

◆ 機能概要

- SpringAOP を利用したトランザクション管理機能である。
- コミット・ロールバックはフレームワークが行なう。
 - AOP を利用したトランザクション制御を行なうため、開発者がトランザクションコードを実装する必要がない。
 - サービス開始時にトランザクションが開始され、終了時にコミットされる
 - 例外発生時、またはユーティリティクラスの呼び出しによりロールバックを行なう。

◆ 概念図



※ 上記の図は TERASOLUNA Server Framework for Java(Web 版) の場合である。
 TERASOLUNA Server Framework for Java(Rich 版) では Action の代わりに Controller が同様

の処理を行っている。

◆ 解説

- AOP を利用したトランザクション制御を行なうため、開発者がトランザクションコードを実装する必要がない。
- サービス層のオブジェクトを境界として、トランザクション制御を行なう。
- コミット・ロールバックはフレームワークが行なう。
 - サービス開始時にトランザクションが開始され、終了時にコミットされる。
 - 例外発生時、またはユーティリティクラスの呼び出しによりロールバックを行なう。
- 指針として、1 サービス・1 トランザクションとなるようにビジネスロジックを実装、設定する。

■ 使用方法

◆ コーディングポイント

- トランザクション伝播の種類
トランザクション伝播には、以下のような種類が存在するが、TERASOLUNA Server Framework for Java では基本的には“REQUIRED”を使用する。必要があれば各自検討し変更すること。

トランザクション伝播	概要
REQUIRED	現在のトランザクションをサポートし、トランザクションが存在しなければ新しいトランザクションを作成する。TERASOLUNA Server Framework for Java で標準的に使用する。
SUPPORTS	現在のトランザクションをサポートし、トランザクションが存在しなければ非トランザクション的に実行する。
MANDATORY	現在のトランザクションをサポートし、トランザクションが存在しなければ例外を送出する。
REQUIRES_NEW	新しいトランザクションを作成し、現在のトランザクションが存在すればそれを一時停止する。
NOT_SUPPORTED	非トランザクション的に実行し、現在のトランザクションが存在すればそれを一時停止する。
NEVER	非トランザクション的に実行し、トランザクションが存在すれば例外を送出する。
NESTED	現在のトランザクションが存在すればネストされたトランザクションの内部で実行し、存在しなければ REQUIRED と同様に動作する。

Spring では、その他にも独立性レベルやタイムアウト、読取専用などの定義情報の設定が可能である。詳細は、SpringAPI を参照のこと。

- コミット、ロールバックの設定

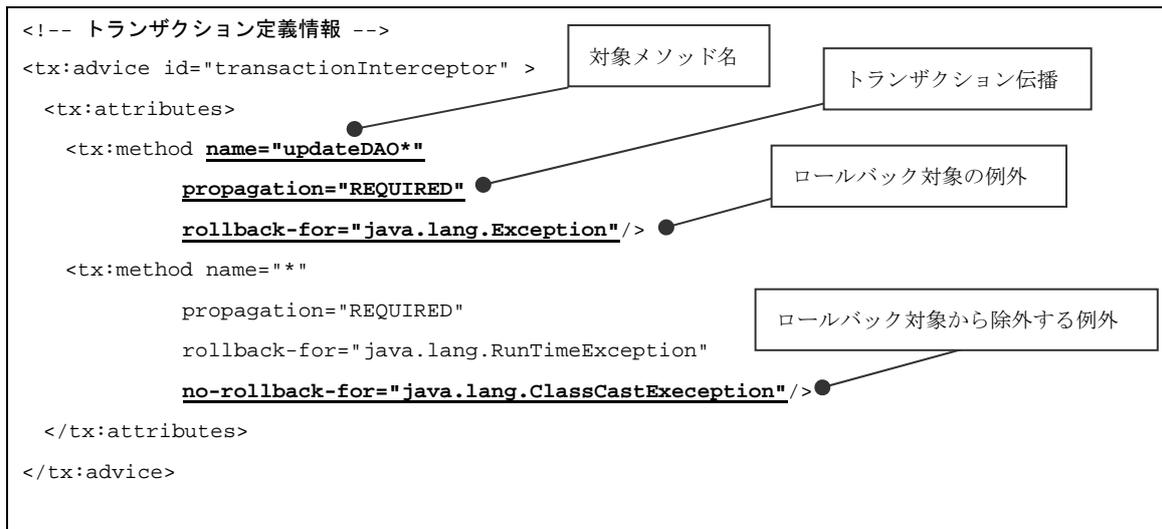
TERASOLUNA Server Framework for Java のトランザクション管理は、Spring の宣言的トランザクション管理機能を利用している。Spring の宣言的トランザクション管理機能は、Bean 定義ファイルの設定をもとに自動的にコミット・ロールバックを行う。開発者がコミット・ロールバック処理を書く必要はない。Spring の宣言的トランザクション管理機能では、基本的にコミット操作が実行され、ロールバック対象となる例外がビジネスロジックによってスローされた場合にロールバックが行われる。ただし検査例外は原則コミットされる。例外をスローせずにロールバックを行いたい場合は後述の TransactionUtil クラスの setRollbackOnly メソッドが利用できる。

また、スローされる例外によってコミット・ロールバックの設定を振りられる。この設定を行うには<tx:method/>要素に以下の属性を追加する。

rollback-for ……指定した例外をロールバック対象とする

no-rollback-for ……指定した例外をロールバック対象から除外する

- コミット・ロールバックの設定を行っている Bean 定義ファイルの実装例



TERASOLUNA Server Framework for Java の推奨設定として、上記の Bean 定義ファイルの実装例のように、「rollback-for="java.lang.Exception"」を設定して、どんな例外がスローされてもロールバックされるようにしている。必要があれば、各プロジェクトでこの設定を変えることもできる。

- AOP を用いたトランザクション設定

AOP によるトランザクション設定を行うには、まずトランザクション処理を行うアドバイス(トランザクションインターセプタ)を定義し、ビジネスロジックのメソッド実行時に AOP を利用してアドバイス(トランザクションインターセプタ)を織り込む。命名規則によって、AOP をかける対象の Bean 定義 ID とメソッド名を指定することで、1 つのトランザクション定義情報を意識することなく複数の Bean で共有することができる。AOP によるトランザクションの詳細については、SpringAPI を参照のこと。

トランザクションマネージャは、Spring が提供する DataSourceTransactionManager を使用する。DataSourceTransactionManager は、単一の JDBC データソースに対してトランザクションを実行するトランザクションマネージャである。

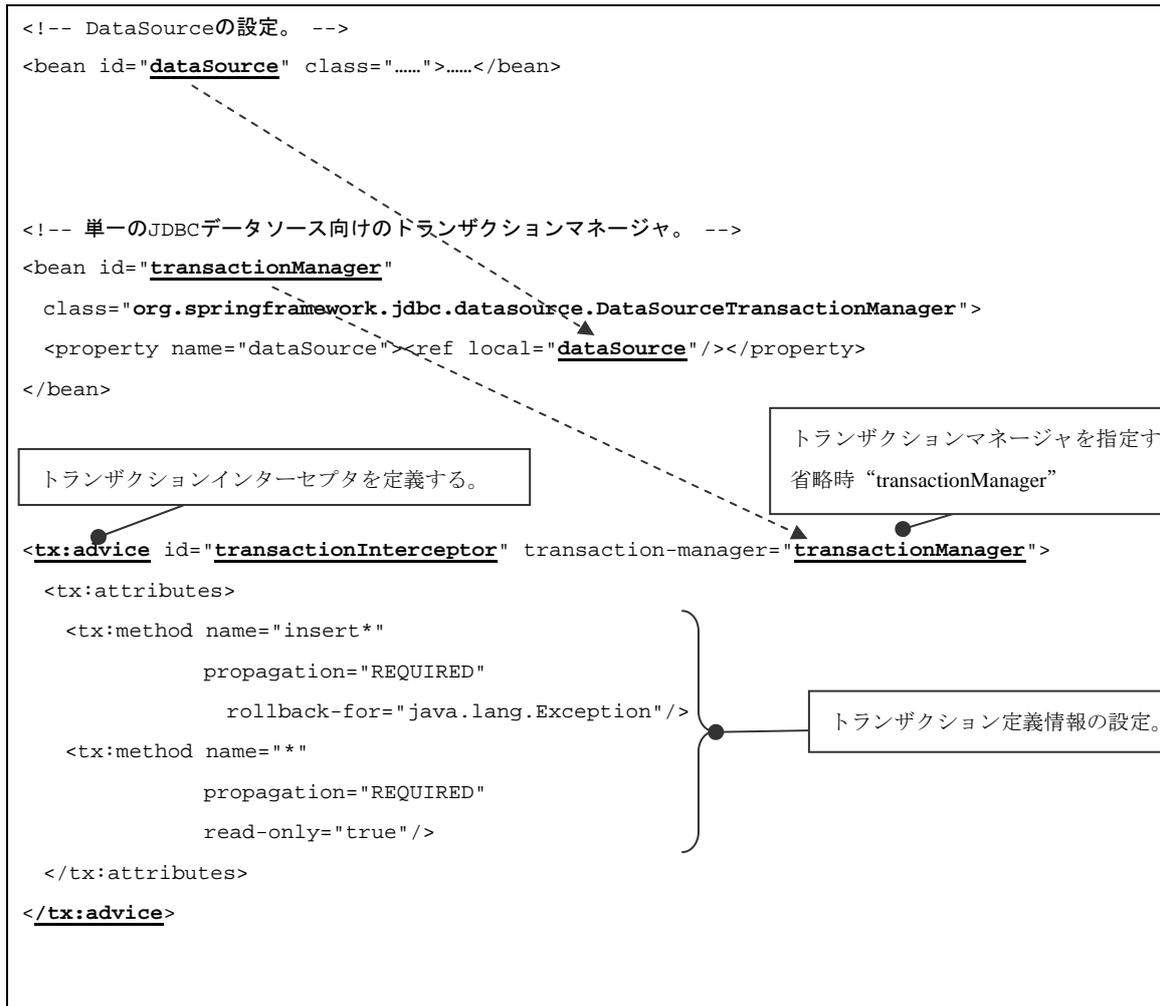
以下に Bean 定義ファイルの設定例を示す。

➤ Bean 定義ファイルの実装例

```
<?xml version="1.0" encoding="UTF-8" ?>
<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:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  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/aop
http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
```

スキーマを定義する。

(続く)



(続く)

```

<!-- AOPの設定 -->
<aop:config>
  <aop:pointcut id="blogicBeans" expression="bean(*BLogic)"/>
  <aop:pointcut id="serviceBeans" expression="bean(*Service)"/>
  <aop:advisor
    pointcut-ref="blogicBeans"
    advice-ref="transactionInterceptor"/>
  <aop:advisor
    pointcut-ref="serviceBeans"
    advice-ref="transactionInterceptor"/>
</aop:config>
<bean id="sumService"
  class="jp.terasoluna.sample2.service.impl.SumServiceImpl"/>

```

AOP の設定を定義する。

AOP でトランザクションをかける対象とする Bean の Bean 定義 ID を指定し、ポイントカット ID で対応付ける。ここでは、Bean 定義 ID の接尾辞が「BLogic」「Service」の Bean に対して適用している。

定義したトランザクションインターセプタを指定する。

各ビジネスロジックの Bean 定義は、命名規則にさえ従えば良い。

- aop:config 定義は、定義した Bean 定義ファイル内の Bean 名に対してのみ有効となるため、Bean 定義ファイルを複数記述する場合は、個々に aop:config 定義を行う必要がある。
以下のように、共通の Bean 定義ファイルを用意して、個々の Bean 定義ファイルから読み込ませることも可能である。

```

<?xml version="1.0" encoding="UTF-8" ?>
<!-- スキーマ定義 (省略) -->
<beans>
  <!-- 共通定義のインポート -->
  <import resource="../commonContext.xml"/>
  <!-- 予約一覧情報を取得するアクション -->
  <bean name="/confirm/reserveList" scope="prototype"
    class="jp.terasoluna.sample.confirm.web.action.ReserveListAction"><!-- オート

```

トランザクション定義などが記述された共通 Bean 定義ファイルをインポートする。

- ロールバックの実装例

ロールバック操作が実行されるには二つのケースがある。ひとつは**実行時例外スロー時にロールバックされるケース。(検査例外はコミットされる)**

もうひとつは、**TransactionUtil** クラスを使用するケースである。

以下にそれぞれのロールバック実装例を示す。

- 例外をスローしロールバックを行うビジネスロジックの実装例

明示的にスローする実装を行っているが、明示的にスローしなくとも、ロールバック設定で指定した例外クラスに基づきフレームワークが自動的にロールバックを行う。

```
public class RunTimeExceptionRollbackBLogicImpl implements BLogic{
    public BLogicResult execute(InputDTO inputDTO) {

        if(業務エラー発生){
            throw new RuntimeException();
        }
    }
}
```

実行時例外をスローする。

- TransactionUtil クラスを使用しロールバックを行うビジネスロジックの実装例

エラー画面に遷移させず、トランザクションの異常をユーザに通知する場合などでは、例外をスローすることが相応しくない場合がある。

例外を発生させなくとも、if 文などの条件分岐でロールバックさせる場合、TransactionUtil クラスの setRollbackOnly メソッドを呼び出すことで、サービス終了時にロールバックさせることができる。

```
import jp.terasoluna.fw.util.TransactionUtil;

public class NoExceptoinRollbackBLogicImpl implements BLogic {
    public BLogicResult execute(InputDTO inputDTO) {
        if(業務エラー発生){
            ///ロールバックを行うためにsetRollbackOnlyメソッドを呼び出す。
            TransactionUtil.setRollbackOnly();
        }
        BLogicResult result = new BLogicResult();
        result.setResultString("failure");
        return result;
    }
}
```

TransactionUtil クラスをインポートする。

setRollbackOnly メソッドを呼び出す。

- 複数 DB 使用時

JTA をサポートする J2EE サーバを使用して、複数 DB に対してトランザクション制御を行なう場合、トランザクションマネージャには、Spring が提供する `JtaTransactionManager` を使用する。`JtaTransactionManager` は、複数のリソースにまたがる可能性のあるトランザクションを実行するトランザクションマネージャであり、JNDI を利用してトランザクションを取得する。データソースはトランザクションマネージャに対して設定するのではなく、個別の DAO に対して、使用するデータソースを設定する。DAO の詳細については、『CB-01 データベースアクセス機能』を参照のこと。

`JtaTransactionManager` の詳細については、SpringAPI を参照のこと。

➤ Bean 定義ファイルの実装例

```
<!-- DataSourceの設定。 -->
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName"><value>DataSource1</value></property>
</bean>
<bean id="dataSource2" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName"><value>DataSource2</value></property>
</bean>
```

transactionManager プロパティに
JtaTransactionManager を指定する。

```
<!-- 複数のデータソース向けのトランザクションマネージャ。 -->
<bean id="transactionManager"
      class="org.springframework.transaction.jta.JtaTransactionManager">
</bean>
```

(トランザクションの設定は、AOPを用いたトランザクション設定と同様)

```
<!-- 業務オブジェクトの定義。トランザクションプロキシにラッピングして定義する。 -->
<bean id="sumService"
      class="jp.terasoluna.sample2.service.impl.SumServiceImpl">
  <property name="queryDAO"><ref bean="queryDAO" /></property>
</bean>
```

```
<!-- DAO定義 -->
<bean id="queryDAO" class="jp.terasoluna.fw.dao.ibatis.QueryDAOiBatisImpl">
  <property name="sqlMapClient"><ref local="sqlMapClient" /></property>
  <property name="dataSource"><ref bean="dataSource" /></property>
</bean>
```

DAO が使用するデータソース
を指定する。

```
<!-- DAO定義 2 -->
<bean id="queryDAO2" class="jp.terasoluna.fw.dao.ibatis.QueryDAOiBatisImpl">
  <property name="sqlMapClient"><ref local="sqlMapClient" /></property>
  <property name="dataSource"><ref bean="dataSource2" /></property>
</bean>
```

- (参考) WebLogic 固有の JTA 設定

Spring では、Spring 標準の JtaTransactionManager と Spring でサポートされている一般的なコンフィグレーションオプションに加えて、WebLogic の JTA 拡張機能を直接利用する特殊な WebLogicJtaTransactionManager を提供している。この WebLogicJtaTransactionManager をトランザクションマネージャに指定することで WebLogic Server の JTA 実装と連携して動作するようにコンフィグレーションすることが可能である。

WebLogicJtaTransactionManager を利用する上でのメリットは以下のようになる。

<メリット>

- ・ エンタープライズアプリケーションの分散トランザクションと 2 フェーズコミットプロトコルを管理できる。
- ・ WebLogic 管理コンソールを通じてトランザクション監視を活用できる。
- ・ データベースごとにアイソレーションレベルを設定することもできるようになるため、複雑なトランザクションコンフィグレーションが可能になる。
- ・ 一時停止されたトランザクションが rollback-only (ロールバックのみ) とマークされているときでも、WebLogic トランザクションを再開することが可能である。ただし、WebLogic の拡張された TransactionManager インタフェースの forceResume() メソッドを呼び出す必要がある。

<補足>

WebLogic 固有の JTA 設定が必要なのは、トランザクション一時停止、または WebLogic の JTA 拡張機能の利用などの上記メリットを実際に活用する場合に限られることに注意が必要である。トランザクション伝播の設定が REQUIRED のような基本的なトランザクション境界設定動作では、標準の JTA 設定で十分である。

➤ WebLogic 固有の JTA 設定を使用した Bean 定義ファイルの実装例

```
<!-- DataSourceの設定。 -->
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName"><value>DataSource1</value></property>
</bean>
<bean id="dataSource2" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName"><value>DataSource2</value></property>
</bean>
<!-- WebLogic固有のJTAトランザクションマネージャ。 -->
<bean id="transactionManager"
  class="org.springframework.transaction.jta.WebLogicJtaTransactionManager">
  <property name="transactionManagerName" value="javax.transaction.TransactionManager"/>
</bean>
```

transactionManager プロパティに WebLogicJtaTransactionManager を指定する。

(トランザクションの設定は、AOPを用いたトランザクション設定と同様)

```
<!-- 業務オブジェクトの定義。トランザクションプロキシにラッピングして定義する。 -->
<bean id="sumService"
  class="jp.terasoluna.sample2.service.impl.SumServiceImpl">
  <property name="queryDAO"><ref bean="queryDAO" /></property>
</bean>
<!-- DAO定義 -->
<bean id="queryDAO" class="jp.terasoluna.fw.dao.ibatis.QueryDAOiBatisImpl">
  <property name="sqlMapClient"><ref local="sqlMapClient" /></property>
  <property name="dataSource"><ref bean="dataSource2" /></property>
</bean>
```

DAO が使用するデータソースを指定する。

● (参考) <tx:method/>の属性一覧

属性	必須	デフォルト	説明
name	○	-	トランザクション対象とするメソッド名 例: 'get*', 'handle*', 'on*Event'
propagation	-	REQUIRED	トランザクション伝播の設定
isolation	-	DEFAULT	トランザクションの分離レベル
timeout	-	-1	トランザクションタイムアウト(秒) 無指定の場合はデフォルトタイムアウト時間が採用される。
read-only	-	false	読み込み専用トランザクションかどうか
rollback-for	-	-	ロールバック対象の例外。 カンマ区切りで複数記述できる。 デフォルトでは <code>java.lang.RuntimeException</code> とその派生クラスが対象となる。
no-rollback-for	-	-	ロールバック対象外の例外。 カンマ区切りで複数記述できる。

◆ 拡張ポイント

なし。

■ 関連機能

- 『CB-01 データベースアクセス機能』
- 『WH-01 ビジネスロジック実行機能』

■ 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」
- 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/commonContext.xml 等
- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC02 トランザクション管理」
 - ◇ /webapps/transaction/*
 - ◇ /webapps/WEB-INF/transaction/*
 - ◇ jp.terasoluna.thin.functionsample.transaction.*

■ 備考

なし

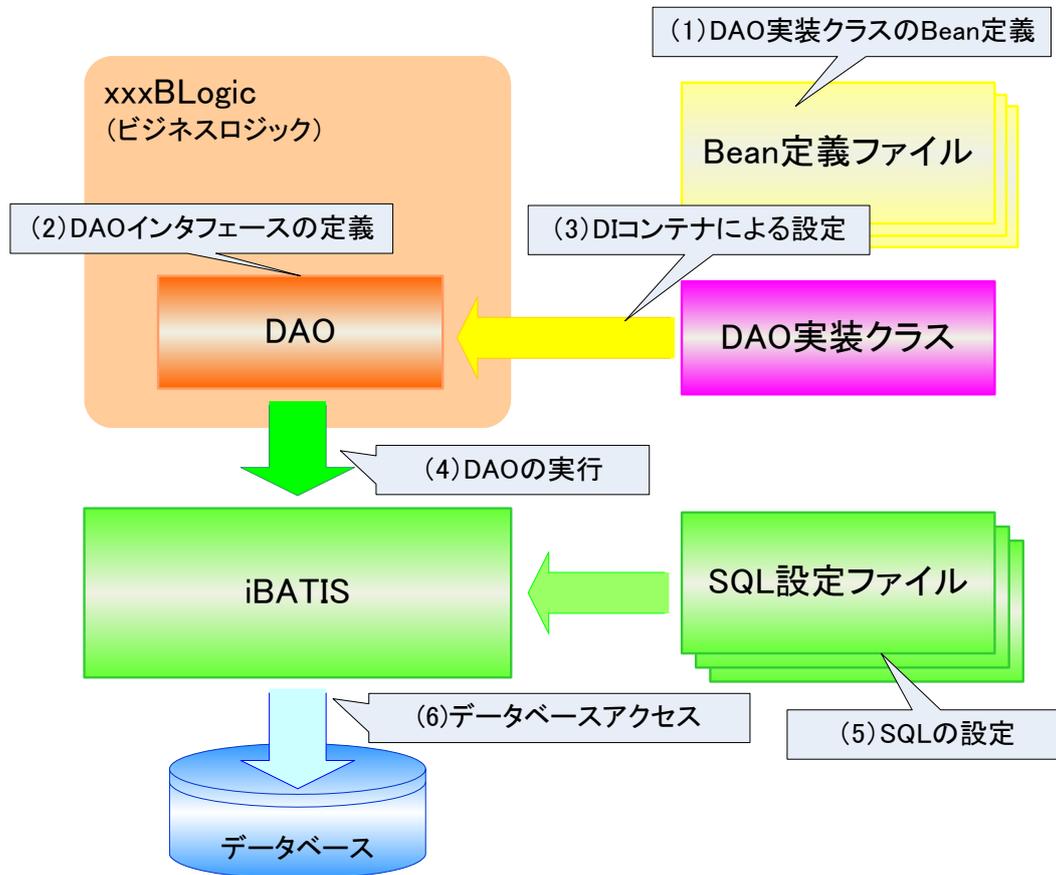
CB-01 データベースアクセス機能

■ 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。
- 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。
- 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 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

```
<!-- (ここでは、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">
```

クラスパスを指定する。

```
<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。

※ 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>要素は maxTransactions 属性、maxSessions 属性の値で同時に SQL を実行する数の制御ができるが、DB 側のロックと組み合わせりデッドロックする場合があるので、アプリケーションサーバのスレッド数より多くしておくこと。なお 512 を超える場合は maxRequests 属性も変更する必要がある。詳しくは iBATIS のドキュメントを参照のこと。

```
<sqlMapConfig>
  <settings
    maxSessions="130"
    maxTransactions="130"
  />
  <sqlMap resource="sqlMap.xml" />
  <sqlMap resource="registerSqlMap.xml" />
</sqlMapConfig>
```

- 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 bean="TerasolunaDataSource" />
  </property>
</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 bean="TerasolunaDataSource"/>
  </property>
</bean>
<bean id="queryDAO"
      class="jp.terasoluna.fw.dao.ibatis.QueryDAOiBatisImpl">
  <property name="sqlMapClient">
    <ref local="sqlMapClient"/>
  </property>
</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 local="sqlMapClient"/>
  </property>
  <property name="dataSource">
    <ref bean="TerasolunaDataSource"/>
  </property>
</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 local="sqlMapClient"/>  
    </property>  
</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 bean="queryDAO"/>
  </property>
</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/oracle10g/101/doc_v5/java.101/B13514-02.pdf

450 ページ「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 と同様のため省略する。

➤ 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 ;
```

◆ 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.ibatis.QueryDAOiBatisImpl	QueryDAO インタフェースの iBATIS 用実装クラス
5	jp.terasoluna.fw.dao.ibatis.UpdateDAOiBatisImpl	UpdateDAO インタフェースの iBATIS 用実装クラス
6	jp.terasoluna.fw.dao.ibatis.StoredProcedureDAOiBatisImpl	StoredProcedureDAO 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

なし。

CC-01 JNDI アクセス機能

■ 概要

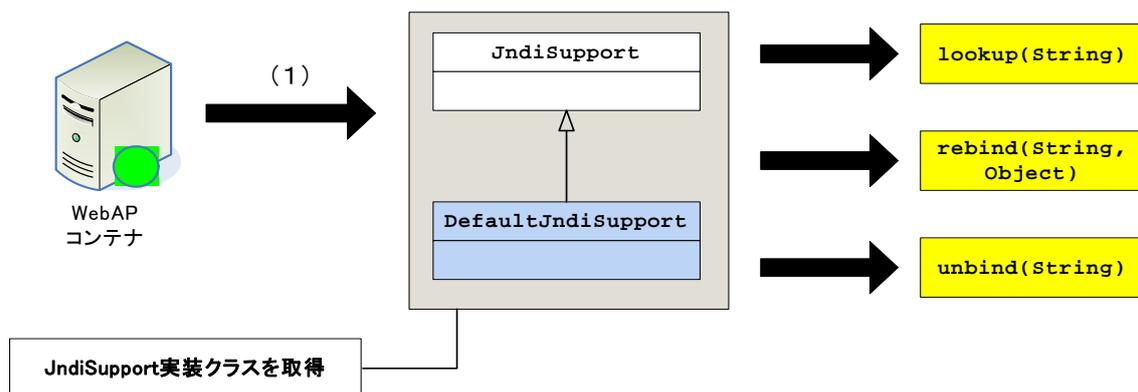
◆ 機能概要

- JNDI 関連機能のサポートインタフェースである。
- WebAP コンテナの JNDI リソースを扱うためにはこのインタフェースを実装する必要がある。
- TERASOLUNA Server Framework for Java ではインタフェースとデフォルト実装クラス DefaultJndiSupport を提供する。

◆ ユーティリティメソッド一覧

メソッド名	概要
lookup(String name)	指定された名前のオブジェクトを取得する
rebind(String name, Object obj)	名前をオブジェクトにバインドして、既存のバインディングを上書きする。
unbind(String name)	指定されたオブジェクトをアンバインドする。

◆ 概念図



◆ 解説

(1) DI コンテナから JndiSupport 実装クラスを取得し、JNDI リソースを利用する。

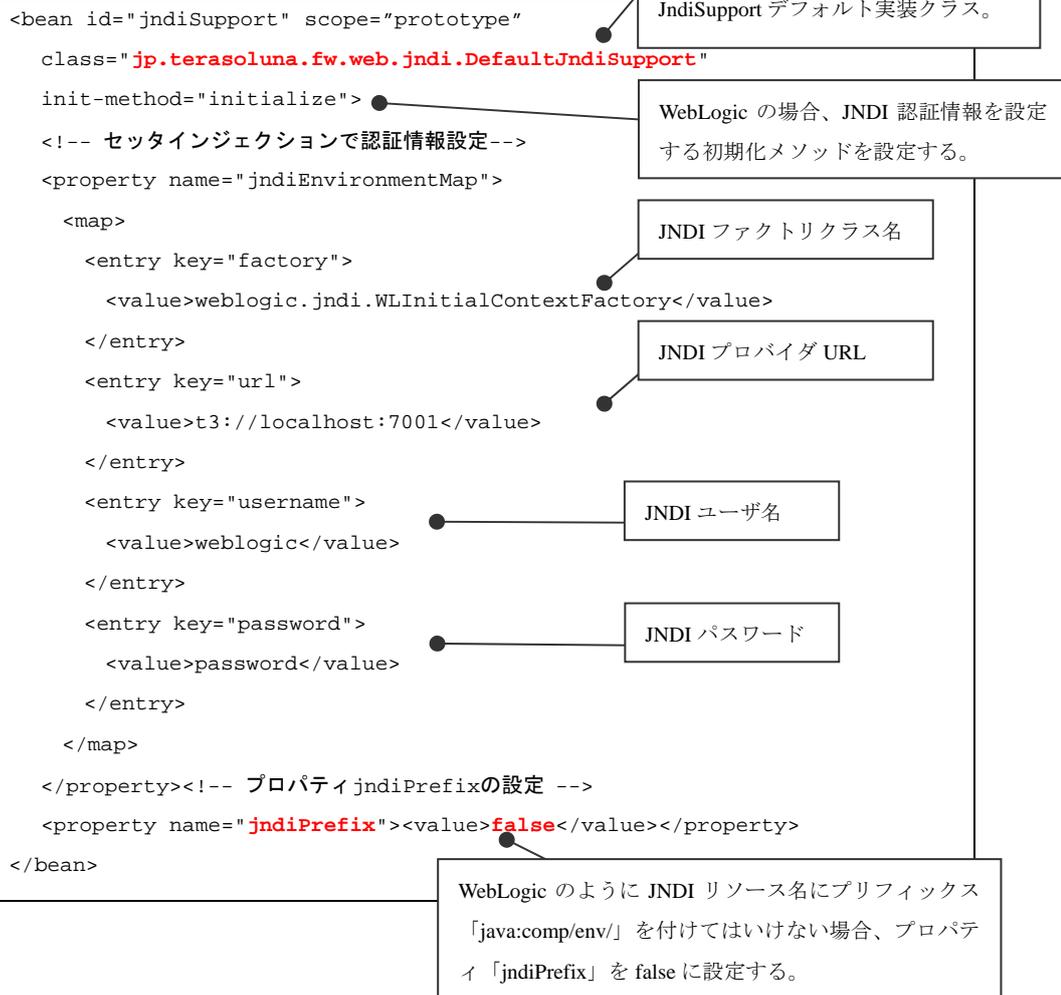
■ 使用方法

◆ コーディングポイント

- JNDI の認証情報が必要な場合は、Bean 定義ファイルに必要なプロパティを以下のように設定する。

設定方法は JNDI サーバの種類によって異なる。

➤ WebLogic の Bean 定義ファイル



➤ Tomcat の Bean 定義ファイル

```
<bean id="jndiSupport" scope="prototype"
  class="jp.terasoluna.fw.web.jndi.DefaultJndiSupport" >
  <!-- プロパティ jndiPrefix の設定 (デフォルト値は false) -->
  <property name="jndiPrefix"><value>false</value></property>
</bean>
```

Tomcat ではこの属性を false にしなければならない。
true に設定すると JNDI リソース名にプリフィックス「java:comp/env/」が付けられ、Tomcat サーバの設定にて登録した read-only のリソースにアクセスできるようになる。Tomcat サーバの設定に登録したリソースにアクセスする必要がある場合は、この設定は false のままで、アクション実装に「java:comp/env/」+JNDI 名のように指定する。

- ビジネスロジックで JNDI リソースを利用するには、以下のような設定を行う。

➤ Bean 定義ファイル

```
<bean id="jndiLogic" scope="prototype"
  class="jp.terasoluna.sample.JndiLogic">
  <property name="jndiSupport">
    <ref bean="jndiSupport" />
  </property>
</bean>
```

JNDI サポートクラス

➤ ビジネスロジック実装例

```
public class JndiLogic {
  private JndiSupport jndiSupport = null;

  public void setJndiSupport(jndiSupport) {
    this.jndiSupport = jndiSupport;
  }

  public Object jndiLookup(String name) {
    return jndiSupport.lookup(name);
  }
}
```

セッターインジェクションで JndiSupport 実装クラスを取得する。

Weblogic の場合、「JNDI リソース名」のみを記述する。

Tomcat の場合、サーバ設定に登録したリソースにアクセスする場合は「java:comp/env/」「JNDI リソース名」を記述する。この場合取得されるリソースは read-only のため、rebind、unbind は使えない。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.jndi.JndiSupport	JNDI 関連のユーティリティインタフェースである。
2	jp.terasoluna.fw.web.jndi.DefaultJndiSupport	TERASOLUNA Server Framework for Java が提供する JNDI 関連のユーティリティデフォルト実装クラスである。
3	jp.terasoluna.fw.web.jndi.JndiException	JNDI 関連エラーを表現する。

◆ 拡張ポイント

JndiSupport インタフェースを実装したクラスを作成し、Bean 定義ファイルに設定を行う。

■ 関連機能

- なし

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC03 JNDI アクセス」
 - ◇ /webapps/jndi/*
 - ◇ /webapps/WEB-INF/jndi/*
 - ◇ jp.terasoluna.thin.functionsample.jndi.*

■ 備考

- なし

CD-01 ユーティリティ機能

■ 概要

◆ 機能概要

- 業務開発で頻繁に使用する処理をユーティリティクラスとして提供する。

◆ 提供ユーティリティクラス一覧

ユーティリティクラス名	概要
jp.terasoluna.fw.util.ClassUtil	文字列(String)から、インスタンスを生成するユーティリティクラス。
jp.terasoluna.fw.util.DateUtil	日付・時刻・カレンダー関連のユーティリティクラス。
jp.terasoluna.fw.util.FileUtil	ファイル操作関連のユーティリティクラス。
jp.terasoluna.fw.util.HashUtil	ハッシュ値を計算するユーティリティクラス。
jp.terasoluna.fw.util.PropertyUtil	プロパティファイルからプロパティを取得するユーティリティクラス。
jp.terasoluna.fw.util.StringUtil	文字列操作を行うユーティリティクラス。

■ ClassUtil

◆ 概要

文字列(String)から、インスタンスを生成するユーティリティクラス。

◆ ユーティリティメソッド一覧

メソッド名	概要
create(String className)	指定されたクラス名を元にインスタンスを生成する。
create(String className, Object[] constructorParameter)	指定されたクラス名を元にインスタンスを生成する。その際、constructorParameter をコンストラクタの引数とする。

◆ 使用方法例

簡単な使用例を以下に示す。

- インスタンスを生成したいクラス (コンストラクタ有り)

```
package jp.terasoluna.xxx;
.....
public class Messages {
    // コンストラクタ。
    public Messages() {
    }
    // コンストラクタ。
    public Messages(Error err) {
        this.add(err);
    }
    .....
}
```

- 使用例

```
// コンストラクタMessages(Error err)を用いて、インスタンスを生成する。
Error err = new Error();
Object[] param = new Object[] { err };
Object obj = ClassUtil.create("jp.terasoluna.xxx.Messages", param);
```

●
コンストラクタ Messages(Error err)を用いて、インスタンスが生成される。

■ DateUtil

◆ 概要

日付・時刻・カレンダー関連のユーティリティクラス。

◆ ユーティリティメソッド一覧

メソッド名	概要
getTimeSystemTime()	システム時刻を取得する。
dateToWarekiString(String format, java.util.Date date)	java.util.Date インスタンスを和暦として、format で指定したフォーマットに変換する。
getWarekiGengoName(Date date)	指定された日付の和暦元号を取得する。
getWarekiGengoRoman(Date date)	指定された日付の和暦元号のローマ字表記(短縮形)を取得する。
getWarekiYear(Date date)	指定された日付の和暦年を取得する。

◆ 使用方法例

和暦関係のメソッドを使用する場合は、`ApplicationResources.properties` に元号名、元号のローマ字表記、元号法施行日を設定する必要がある。

● `ApplicationResources.properties` の設定例

```
wareki.gengo.0.name=平成
wareki.gengo.0.roman=H
wareki.gengo.0.startDate=1989/01/08
wareki.gengo.1.name=昭和
wareki.gengo.1.roman=S
wareki.gengo.1.startDate=1926/12/25
wareki.gengo.2.name=大正
.....
```

元号名

元号のローマ字表記

元号法施行日（西暦:yyyy/MM/dd 形式）

上記 3 つの設定の関連付けにのみ使用する。任意の文字列を設定できる。

● 使用例

```
// 引数を準備
df = new SimpleDateFormat("yyyy.MM.dd HH:mm:ss");
date = new Date(df.parse("1978.01.15 00:00:00").getTime());
```

```
// 和暦フォーマットに変換する。（結果：wareki = "昭和53年01月15日日曜日"）
String wareki = DateUtil.dateToWarekiString("GGGGyy年MM月dd日EEEE",
date);
```

「G」:元号を表すパターン文字。

例：

(4 個以上の連続したパターン文字)

明治、大正、昭和、平成

(3 個以下の連続したパターン文字)

M、T、S、H

「y」:年（和暦）を表すパターン文字。

「E」:曜日を表すパターン文字。

例：

(4 個以上の連続したパターン文字)

月曜日、火曜日、水曜日

(3 個以下の連続したパターン文字)

月、火、水

```
// 元号を取得する。（結果：gengo = "昭和"）
String gengo = DateUtil.getWarekiGengoName(date);
```

```
// 元号（ローマ字表記）を取得する。（結果：gengoR = "S"）
String gengoR = DateUtil.getWarekiGengoRoman(date);
```

```
// 和暦年を取得する。（結果：year = "53"）
String year = DateUtil.getWarekiYear(date);
```

■ FileUtil

◆ 概要

ファイル操作関連のユーティリティクラス。このユーティリティクラスにて、「セッション ID に対応するディレクトリ」とは、セッション ID の 16 進ハッシュ値をディレクトリ名とするディレクトリのことを意味する。

※`rmdir(File dir)`メソッドについては、セッション ID とは関連しないメソッドなので、注意すること。

◆ ユーティリティメソッド一覧

メソッド名	概要
<code>getSessionDirectoryName(String sessionId)</code>	指定されたセッション ID に対応するディレクトリ名を取得する。
<code>getSessionDirectory(String sessionId)</code>	指定されたセッション ID に対応するディレクトリをベースディレクトリから取得する。 プロパティにてベースディレクトリを設定を行なわなかった、またはプロパティ値が空文字の場合、ベースディレクトリとして <code>temp</code> ディレクトリを用いる。
<code>makeSessionDirectory(String sessionId)</code>	指定されたセッション ID に対応するディレクトリを作成する。 作成が成功した場合には、 <code>true</code> を返す。
<code>removeSessionDirectory(String sessionId)</code>	指定されたセッション ID に対応するディレクトリを削除する。 削除が成功した場合には、 <code>true</code> を返す。
<code>rmdir(File dir)</code>	指定されたディレクトリを削除する。 ディレクトリ内にファイル、ディレクトリがある場合でも、再帰的に削除される。

◆ 使用方法例

システム設定プロパティファイル (`system.properties`) にセッションディレクトリのベースとなるパスを設定することで、そのパスの配下のディレクトリに対して処理を行なう。プロパティが設定されていない場合は、`temp` ディレクトリ (`/temp`) を用いる。

- システム設定プロパティファイルの設定例

```
session.dir.base=/tmp/sessions
```

ベースディレクトリのパスを設定する。

● 使用例

```
// 指定したセッションIDに対応するディレクトリ名を取得する。
// (結果: dirNameは、指定したセッションIDに対応するディレクトリ名。)
String dirName = FileUtil.getSessionDirectoryName("0123abc");

// 指定したセッションIDに対応するディレクトリを取得する。
// (結果: dirには、/tmp/sessions配下の
// セッションIDに対応するディレクトリが格納される。)
File dir = FileUtil.getSessionDirectory("0123abc");

// 指定したセッションIDに対応するディレクトリを作成する。
// (結果: /tmp/sessions配下にセッションIDに対応するディレクトリが作成される。)
boolean makeResult = FileUtil.makeSessionDirectory("0123abc");

// 指定したセッションIDに対応するディレクトリを削除する。
// (結果: /tmp/sessions配下のセッションIDに対応するディレクトリが削除される。)
boolean removeResult = FileUtil.removeSessionDirectory("0123abc");

// 指定したディレクトリを削除する。
// (結果: ディレクトリ/rmtemp/rmdirs1が削除される。)
File dir = new File("/rmtemp/" + "rmdirs1");
boolean result = FileUtil.rmdir(dir);
```

セッション ID に対応するディレクトリ
="0123abc"の16進ハッシュ値

■ HashUtil

◆ 概要

ハッシュ値を計算するユーティリティクラス。

◆ ユーティリティメソッド一覧

メソッド名	概要
hash(String algorithm, String str)	指定されたアルゴリズムで文字列のハッシュ値を取得する。
hashMD5(String str)	MD5 アルゴリズムで文字列のハッシュ値を取得する。
hashSHA1(String str)	SHA1 アルゴリズムで文字列のハッシュ値を取得する。

◆ 使用方法例

簡単な使用例を以下に示す。使い方が容易なメソッドについては省略する。

```
// ハッシュ値を取得する。  
// (結果: hashValueには、  
// MessageDigest.getInstance("MD5").digest("abc".getBytes())  
// と等しい値が格納される。)  
byte[] hashValue = HashUtil.hash(paramAlgorithm, paramStr);
```

■ PropertyUtil

◆ 概要

プロパティファイルからプロパティを取得するユーティリティクラス。

◆ ユーティリティメソッド一覧

メソッド名	概要
addPropertyFile(String name)	指定されたプロパティファイルを追加で読み込む。
getProperty(String key)	指定されたキーのプロパティを取得する。
getProperty(String key, String defaultValue)	指定されたキーのプロパティを取得する。 プロパティが見つからなかった場合には、指定されたデフォルトが返される。
getPropertyNames()	プロパティのすべてのキーのリストを取得する。
getPropertyNames(String keyPrefix)	指定されたプリフィックスから始まるキーのリストを取得する。
getPropertyValues(String propertyName, String keyPrefix)	プロパティファイル名、部分キー文字列を指定することにより値セットを取得する。
getPropertyNames(Properties localProps, String keyPrefix)	プロパティを指定し、部分キープリフィックスに合致するキー一覧を取得する。
getPropertyValues(Properties localProps, Enumeration<String> propertyNames)	キー一覧に対し、プロパティより取得した値を取得する。
loadProperties(String propertyName)	指定したプロパティファイル名で、プロパティオブジェクトを取得する。

◆ 使用方法例

各メソッドの簡単な使用例を以下に示す。提供するメソッドは、主に、プロパティの読み込み、読み込み済みプロパティの取得、プロパティオブジェクトに対する操作の3通りに分類される。

PropertyUtil は、クラスパス上の「ApplicationResources.properties」というファイルからプロパティの読み込みを行う。また、プロパティファイルに他のファイル名を指定することにより、他のプロパティファイルを追加で読み込むことができる。

※ プロパティファイルに同一のキーのプロパティが重複して存在していた場合、後から読み込まれた設定の値が有効になる。

- プロパティファイル追加の指定方法

```
add.property.file.<1からの連番> = <追加したいプロパティファイル名>
```

- プロパティファイル(ApplicationResources.properties)記述例

```
add.property.file.1=error.properties  
add.property.file.2=xxxxxx.properties  
add.property.file.3=yyyyyy.properties
```

- Application.properties の記述により追加で読み込まれたプロパティファイル (errors.properties)の内容

```
error.message.01=エラーメッセージ1  
error.message.02=エラーメッセージ2
```

また、以下のように PropertyUtil のメソッドを利用してプロパティファイルを追加で読み込むこともできる。

- PropertyUtil のメソッドを利用して追加で読み込むプロパティファイル (test.properties)の内容

```
test.message.01=テストメッセージ1  
test.message.02=テストメッセージ2  
error.message.03=エラーメッセージ3
```

- プロパティファイルの読み込み

```
// プロパティファイルを追加で読み込む .propertiesが付いていても付いてなくてもOK
PropertyUtil.addPropertyFile("test");
```

- プロパティの取得

```
// test.message.01のプロパティ値を取得する。(結果: str = "テストメッセージ1")
String str1 = PropertyUtil.getProperty("test.message.01");

// error.message.04のプロパティ値を取得する。
// プロパティ設定がない場合はデフォルトのメッセージを取得する。
// (結果: str2 = "デフォルト")
String str2 = PropertyUtil.getProperty("error.message.04", "デフォルト");

// プロパティの全てのキーのリストを取得する。
// (結果: en1には、error.message.01~03、test.message.01~02が格納される。)
Enumeration en1 = PropertyUtil.getPropertyNames();

// error.messageで始まるプロパティのキーのリストを取得する。
// (結果: en2にはerror.message.01~03が格納される。)
Enumeration en2 = PropertyUtil.getPropertyNames("test.message");

// error.properties内の、errorで始まるキーのプロパティ値を取得する。
// (結果: set1には"エラーメッセージ1"、"エラーメッセージ2"が格納される。)
Set set1 = PropertyUtil.getPropertiesValues("error", "error");
```

- プロパティオブジェクトに対する操作

```
// test.propertiesのPropertyオブジェクトを取得する。
// (結果: test.propertiesをロードした場合のPropertyオブジェクトが格納される。)
Properties prop = PropertyUtil.loadProperties("test");

// propプロパティオブジェクト内の、errorで始まるキーのリストを取得する。
// (結果: en3にはerror.message.03が格納される。)
Enumeration en3 = PropertyUtil.getPropertyNames(prop, "error");

// propプロパティオブジェクト内の、error、testで始まるキーのプロパティ値を取得する。
// (結果: set2には"テストメッセージ1"、"テストメッセージ2"、
// "エラーメッセージ3が格納される")
Enumeration em = new StringTokenizer("test error");
Set set2 = PropertyUtil.getPropertiesValues(prop, em);
```

■ StringUtil

◆ 概要

文字列操作を行なうユーティリティクラス。

◆ ユーティリティメソッド一覧

メソッド名	概要
isWhitespace(char c)	指定された文字がホワイトスペースかどうかを判別する。
rtrim(String str)	文字列の右側のホワイトスペースを削除する。 引数が <code>null</code> のときは <code>null</code> を返す。
ltrim(String str)	文字列の左側のホワイトスペースを削除する。 引数が <code>null</code> のときは <code>null</code> を返す。
trim(String str)	文字列の両側のホワイトスペースを削除する。 引数が <code>null</code> のときは <code>null</code> を返す。
toShortClassName(String longClassName)	完全修飾クラス名から短縮クラス名 (パッケージ修飾なし) を取得する。
getExtension(String name)	指定された文字列から末尾の拡張子を取得する。 拡張子がない場合は空文字列を返す。
toHexString(byte[] byteArray, String delimiter)	バイト配列を 16 進文字列に変換する。
capitalizeInitial(String str)	指定された文字列の頭文字を大文字にする。
parseCSV(String csvString)	CSV 形式の文字列を文字列の配列に変換する。 文字列の先頭がカンマで始まっていたり、文字列の最後がカンマで終わっている場合には、それぞれ変換結果の配列の最初か、あるいは最後の要素が空文字列となるように変換される。 カンマが連続している場合には、空文字列として変換される。 <code>csvString</code> が <code>null</code> だった場合には、要素数 0 の配列に変換される。
parseCSV(String csvString, String escapeString)	CSV 形式の文字列を文字列の配列に変換する。 文字列の先頭がカンマで始まっていたり、文字列の最後がカンマで終わっている場合には、それぞれ変換結果の配列の最初か、あるいは最後の要素が空文字列となるように変換される。 カンマが連続している場合には、空文字列として変換される。 <code>csvString</code> が <code>null</code> だった場合には、要素数 0 の配列に変換される。 エスケープ文字列に設

	定された文字列の次にあるカンマは区切り文字としては認識しない。 エスケープ文字列の後のエスケープ文字列はエスケープ文字として認識しない。
<code>dump(Map<?, ?> map)</code>	引数のマップのダンプを取得する。 値オブジェクトに配列が含まれている場合、各要素オブジェクトの <code>toString()</code> を行い、文字列を出力する。
<code>getArraysStr(Object[] arrayObj)</code>	ダンプ対象の値オブジェクトが配列形式の場合、配列要素でなくなるまで繰り返し値を取得する。
<code>hankakuToZenkaku(String value)</code>	半角文字列を全角文字列に変換する。 カナ文字に濁点または半濁点が続く場合は、可能な限り1文字に変換する。
<code>zenkakuToHankaku(String value)</code>	全角文字列を半角文字列に変換する。
<code>filter(String str)</code>	HTML メタ文字列に変換する。
<code>toLikeCondition(String condition)</code>	検索条件文字列を LIKE 述語のパターン文字列に変換する。
<code>getByteLength(String value, String encoding)</code>	指定された文字列のバイト列長を取得する。

◆ 使用方法例

各メソッドの簡単な使用例を以下に示す。使い方が容易なメソッドについては省略する。

```
// 短縮クラス名を取得する。(結果: shortClassName = "String")
String shortClassName = toShortClassName("java.lang.String");

// 拡張子を取得する。(結果: extention1 = ".txt", extention2 = ".xls")
String extention1 = getExtention("Test.txt");
String extention2 = getExtention("Test1.Test2.xls");

// バイト配列を16進文字列に変換する。(結果: hexString = "00/0A/64")
byte[] byteArray = {0, 10, 100};
String hexString = StringUtil.toHexString(byteArray, "/");

// CSVをString配列に変換する。
// (結果: strArray = {"bc", "def", "ghi", "jk,l"})
String[] strArray = parseCVS("abc,def,ghi,jk/,l", "/");
```

(続く)

```
// 入出力マップのダンプを取得する。(結果:dump = "String" となる。
Map<String, String> map = new LinkedHashMap<String, String>();
map.put("1","東京");
.....
String dumpStr = dump(map);

// ダンプ用に配列要素でなくなるまで繰り返し値を取得する。
// (結果:arrayStr = "{1,2,3,4,5}")
String[] str = {"1", "2", "3", "4", "5"};
String arrayStr = StringUtil.getArraysStr(str);

// 半角文字列を全角文字列に変換する。(結果:zenkaku = "ア`!A8")
String zenkaku = StringUtil.hankakuToZenkaku("ア`!A8");

// 全角文字列を半角文字列に変換する。(結果:hankaku = "A!7")
String hankaku = StringUtil.zenkakuToHankaku("A!ア");

// HTMLメタ文字列変換を行なう。
// (結果:meta = "&lt; &amp; &gt; &quot; ア")
String meta = StringUtil.filter("< & > ¥" ア");

// LIKE述語のパターン文字列に変換する。(結果:like = "~_a~%%")
String like = StringUtil.toLikeCondition("_a%");

// バイト列長を取得する。(結果:i = 9)
int i = StringUtil.getBytesLength("あああ", "UTF-8");
```

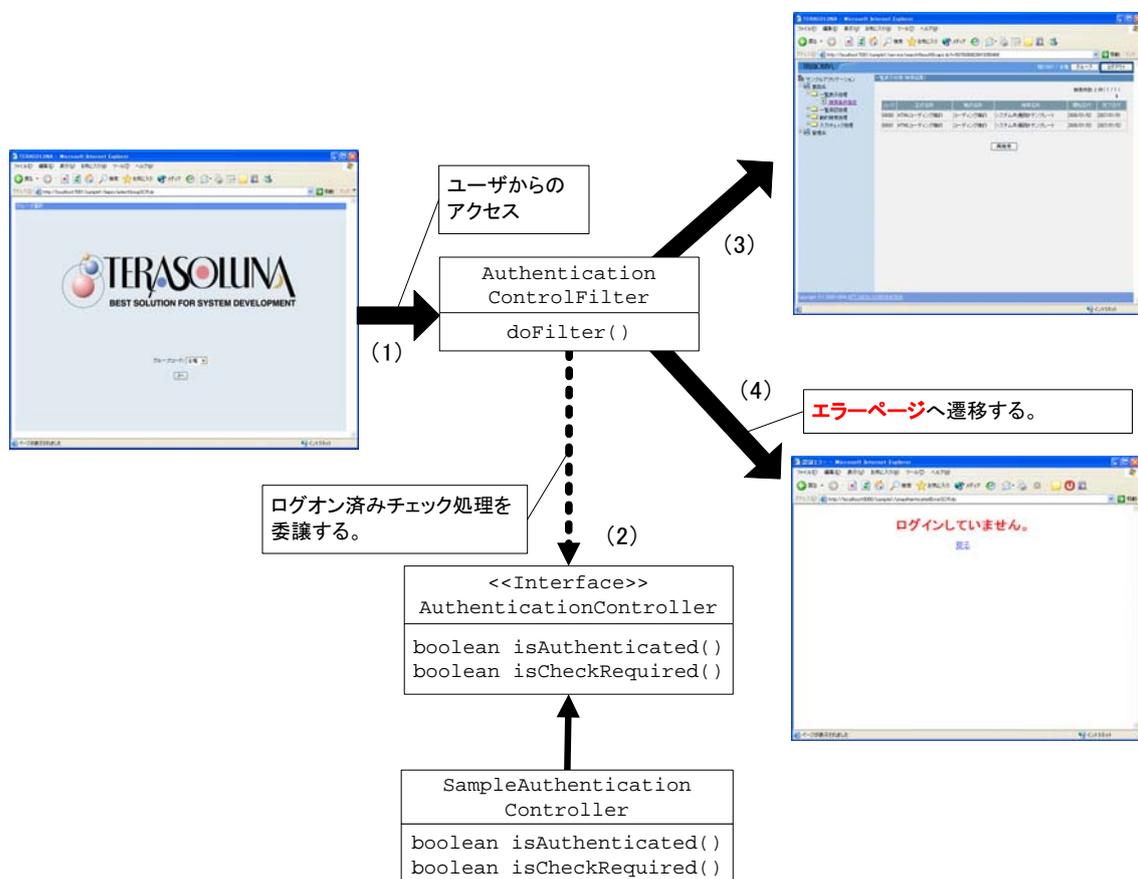
WA-01 ログオン済みチェック機能

■ 概要

◆ 機能概要

- ログオン済みユーザからのアクセスであるかをチェックする。ログオン済みではない場合は、アクセスをブロックし、エラーページへ遷移する。

◆ 概念図



◆ 解説

以下の解説はデプロイメントディスクリプタ（web.xml）に `AuthenticationControlFilter` の設定が正しくされている事が前提となる。設定方法の詳細は、コーディングポイントを参照のこと。

- (1) WebAP コンテナがユーザからのアクセスを受け `AuthenticationControlFilter` に処理を

委譲する。

- (2) `AuthenticationControlFilter` はデプロイメントディスクリプタ (`web.xml`) の設定に従い、任意の `AuthenticationController` インタフェース実装クラスにアクセス権限チェック処理を委譲する。
- (3) ユーザがログオン済みだった場合、次の画面に遷移する。
- (4) ユーザがログオン済みでなかった場合、デプロイメントディスクリプタ (`web.xml`) の設定に従いエラーページへ遷移する。

■ 使用方法

◆ コーディングポイント

ログオン済みチェック機能は、サーブレットフィルタを用いて実現している。フィルタ内部から DI コンテナに対して、ログオン済みチェックの実装された `Bean` を取得するが、その際取得する `Bean` の `id` 属性についてデフォルトで用意したものを取得するか、指定したものを使うか選択できる。

以下では `Bean` 定義ファイルにログオン済みチェックを実装した `Bean` を定義する際の `id` 属性を、デフォルトで用意されているものを使う場合と、`id` 属性を指定して使う場合の 2 例について設定方法を説明する。

ログオン済みチェックの実装についてはどちらの場合も同じため、最後に 1 例で説明する。

なお、『WA-02 アクセス権限チェック機能』、『WA-03 サーバー閉塞チェック機能』、および『WA-04 業務閉塞チェック機能』も同様の方法で実現している。

1. デフォルトの id 属性の値を使う場合の設定方法

● デプロイメントディスクリプタ (web.xml)

```
<webapp>
.....
<filter>
  <filter-name>authenticationControlFilter</filter-name>
  <filter-class>
    jp.terasoluna.fw.web.thin.AuthenticationControlFilter
  </filter-class>
</filter>
.....
<filter-mapping>
  <filter-name>authenticationControlFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
.....
<error-page>
  <exception-type>
    jp.terasoluna.fw.web.thin.UnauthenticatedException
  </exception-type>
  <location>/authenticatedError.jsp</location>
</error-page>
.....
</web-app>
```

同名で指定する。
任意の名前でよい。

AuthenticationControlFilter を設定する。

フィルタを動作させたい
リクエストパスを書く。

UnauthenticatedException を設定する。

ログオン済みでなかった場合に
遷移させるページを設定する。

● Bean 定義ファイル

```
<beans>
.....
<bean id="authenticationController"
  class="jp.terasoluna.sample.SampleAuthenticationController"/>
.....
</beans>
```

デフォルトの id 属性の値。AuthenticationControlFilter の場合は authenticationController を設定する。

AuthenticationController の実装クラスを設定する。

2. id 属性の値を変えて使う場合の設定方法

● デプロイメントディスクリプタ (web.xml)

```
<web-app>
<filter>
.....
<filter-name>authenticationControlFilter</filter-name>
<filter-class>
  jp.terasoluna.fw.web.thin.AuthenticationControlFilter
</filter-class>
<init-param>
  <param-name>controller</param-name>
  <param-value>
    sampleAuthenticationController
  </param-value>
</init-param>
</filter>
.....
<filter-mapping>
  <filter-name>authenticationControlFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
.....
<error-page>
  <exception-type>
    jp.terasoluna.fw.web.thin.UnauthenticatedException
  </exception-type>
  <location>/authenticatedError.jsp</location>
</error-page>
.....
</web-app>
```

AuthenticationControlFilter を設定する。

パラメータ名は controller と設定する。

同名で指定する。任意の名前でよい。

フィルタを動作させたいリクエストパスを書く。

UnauthenticatedException を設定する。

同名で指定する。任意の名前でよい。

ログオン済みでなかった場合に遷移させるページを設定する。

● bean 定義ファイル

```
<beans>
.....
<bean id="sampleAuthenticationController"
  class="jp.sample.SampleAuthenticationController" />
.....
</beans>
```

AuthenticationController の実装クラスを設定する。

※コントローラの bean 定義は必ず applicationContext.xml に記述する必要がある。

3. AuthenticationController の実装クラス例

```
package jp.terasoluna.sample;

import javax.servlet.ServletException;
import jp.terasoluna.fw.web.thin.AuthenticationController;

public class SampleAuthenticationController
    implements AuthenticationController {

    // ログオン済みチェックが必要ないパス
    private String noCheckRequiredPath = "/sample/index.jsp";

    // アクセス権限がある場合はtrueを返し、ない場合はfalseを返す
    public boolean isAuthenticated(String pathInfo,
        ServletRequest req) {

        // セッションを取得
        HttpSession session = ((HttpServletRequest) req).getSession();

        // セッションからUserValueObjectを取得
        SampleUserValueObject uvo = (SampleUserValueObject)
            session.getAttribute(UserValueObject.USER_VALUE_OBJECT_KEY);

        // 例はログオンした場合のみセッションにUserValueObjectを格納するという前提なので
        // UserValueObjectがセッションにあった場合はログオン済みとする
        if (uvo != null) {
            return true;
        }
        return false;
    }

    // ログオン済みチェックが必要ならtrue、必要ないならfalseを返す
    public boolean isCheckRequired(ServletRequest req) {
        if (noCheckRequiredPath.equals(RequestUtil.getPathInfo(req))) {
            return false;
        }
        return true;
    }
}
```

AuthenticationController インタフェースを実装する。

引数 pathInfo は RequestUtil.getPathInfo(req)と同等の値が設定されているとは限らないので注意すること。RequestUtilについては『CD-04 ユーティリティ機能』を参照のこと。

※この実装例は簡単な説明のため、例外の発生を考慮していないことに注意すること。

※AuthenticationController 実装クラスはスレッドセーフに実装する必要があるが、この例では必要ないため特別なことをしていないことに注意すること。

◆ 拡張ポイント

ログオン済みチェック機能は、**AuthenticationController** インタフェースを実装して利用する。コーディングポイントの説明例を参照のこと。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.thin. AuthenticationControlFilter	ログオン済みのユーザのみ許可するように制御するフィルタクラス。
2	jp.terasoluna.fw.web.thin. AuthenticationnController	AuthenticationControlFilter から呼び出されるクラスは、このインタフェースを実装する必要がある。
3	jp.terasoluna.fw.web.thin. UnauthenticatedException	ログオン済みでないユーザがアクセスしたことを通知するための例外クラス。

■ 関連機能

- 『CD-04 ユーティリティ機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) チュートリアル
 - 「2.10 アクセス制御」
 - /webapps/WEB-INF/applicationContext.xml
 - /webapps/WEB-INF/web.xml
 - jp.terasoluna.thin.tutorial.web.SampleAuthController
- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC04 ログオン済みチェック」
 - ◇ /webapps/authentication/*
 - ◇ /webapps/WEB-INF/authentication/*
 - ◇ jp.terasoluna.thin.functionsample.authentication.*

■ 備考

- なし

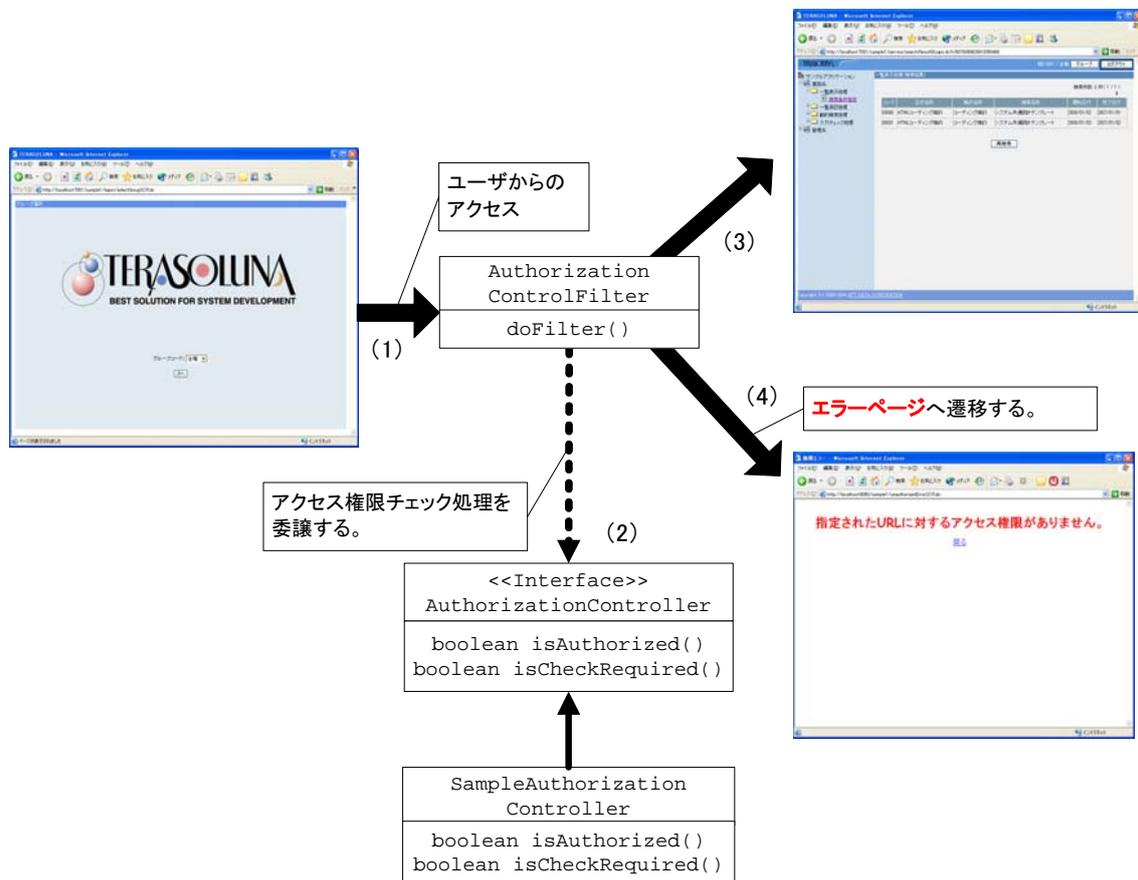
WA-02 アクセス権限チェック機能

■ 概要

◆ 機能概要

- アクセス権限を持つユーザからのアクセスであるかをチェックする。アクセス権限を持つユーザではない場合は、アクセスをブロックし、エラーページへ遷移する。

◆ 概念図



◆ 解説

以下の解説はデプロイメントディスクリプタ（web.xml）に `AuthorizationControlFilter` の設定が正しくされている事が前提となる。設定方法の詳細は、コーディングポイントを参照のこと。

- (1) WebAP コンテナがユーザからのアクセスを受け `AuthorizationControlFilter` に処理を委譲する。
- (2) `AuthorizationControlFilter` はデプロイメントディスクリプタ (`web.xml`) の設定に従い、任意の `AuthorizationController` インタフェース実装クラスにアクセス権限チェック処理を委譲する。
- (3) ユーザがアクセス権限を持っていた場合、次の画面に遷移する。
- (4) ユーザがアクセス権限を持っていなかった場合、デプロイメントディスクリプタ (`web.xml`) の設定に従いエラーページへ遷移する。

■ 使用方法

◆ コーディングポイント

アクセス権限チェック機能は、サーブレットフィルタを用いて実現している。フィルタ内部から DI コンテナに対して、アクセス権限チェックの実装された `Bean` を取得するが、その際取得する `Bean` の `id` 属性についてデフォルトで用意したものを取得するか、指定したものを使うか選択できる。

以下では `Bean` 定義ファイルにアクセス権限チェックを実装した `Bean` を定義する際の `id` 属性を、デフォルトで用意されているものを使う場合と、`id` 属性を指定して使う場合の 2 例について設定方法を説明する。

アクセス権限チェックの実装についてはどちらの場合も同じたため、最後に 1 例で説明する。

なお、『WA-01 ログオン済みチェック機能』、『WA-03 サーバー閉塞チェック機能』、および『WA-04 業務閉塞チェック機能』も同様の方法で実現している。

1. デフォルトの id 属性の値を使う場合の設定方法

● デプロイメントディスクリプタ (web.xml)

```
<web-app>
.....
<filter>
  <filter-name>authorizationControlFilter</filter-name>
  <filter-class>
    jp.terasoluna.fw.web.thin.AuthorizationControlFilter
  </filter-class>
</filter>
.....
<filter-mapping>
  <filter-name>authorizationControlFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
.....
<error-page>
  <exception-type>
    jp.terasoluna.fw.web.thin.UnauthorizedException
  </exception-type>
  <location>/authorizedError.jsp</location>
</error-page>
.....
</web-app>
```

同名で指定する。
任意の名前でよい。

AuthorizationControlFilter を設定する。

フィルタを動作させたい
リクエストパスを書く。

UnauthorizedException を設定する。

アクセス権限がなかった場合に
遷移させるページを設定する。

● Bean 定義ファイル

```
<beans>
.....
<bean id="authorizationController"
  class="jp.terasoluna.sample.SampleAuthorizationController"/>
.....
</beans>
```

デフォルトの id 属性の値。AuthorizationControlFilter の場合は
authorizationController を設定する。

AuthorizationController の実装クラスを設定する。

2. id 属性の値を変えて使う場合の設定方法

● デプロイメントディスクリプタ (web.xml)

```

<web-app>
  .....
  <filter>
    <filter-name>authorizationControlFilter</filter-name>
    <filter-class>
      jp.terasoluna.fw.web.thin.AuthorizationControlFilter
    </filter-class>
    <init-param>
      <param-name>controller</param-name>
      <param-value>
        sampleAuthorizationController
      </param-value>
    </init-param>
  </filter>
  .....
  <filter-mapping>
    <filter-name>authorizationControlFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  .....
  <error-page>
    <exception-type>
      jp.terasoluna.fw.web.thin.UnauthorizedException
    </exception-type>
    <location>/authorizedError.jsp</location>
  </error-page>
  .....
</web-app>

```

AuthorizationControlFilter を設定する。

パラメータ名は controller と設定する。

同名で指定する。任意の名前でよい。

フィルタを動作させたいリクエストパスを書く。

同名で指定する。任意の名前でよい。

UnauthorizedException を設定する。

アクセス権限がなかった場合に遷移させるページを設定する。

● bean 定義ファイル

```

<beans>
  .....
  <bean id="sampleAuthorizationController"
    class="jp.terasoluna.sample.SampleAuthorizationController"/>
  .....
</beans>

```

AuthorizationController の実装クラスを設定する。

※コントローラの bean 定義は必ず applicationContext.xml に記述する必要がある。

3. AuthorizationController の実装クラス例

```
package jp.terasoluna.sample;

import javax.servlet.ServletException;
import jp.terasoluna.fw.web.thin.AuthorizationController;

public class SampleAuthorizationController
    implements AuthorizationController {

    // アクセス権限チェックが必要なパス
    private String checkRequiredPath = "/sample/admin.jsp";

    // アクセス権限がある場合はtrueを返し、ない場合はfalseを返す
    public boolean isAuthorized(String pathInfo,
                               ServletRequest req) {

        // セッションを取得
        HttpSession session = ((HttpServletRequest) req).getSession();

        // セッションからUserValueObjectを取得
        SampleUserValueObject uvo = (SampleUserValueObject)
            session.getAttribute(UserValueObject.USER_VALUE_OBJECT_KEY);

        // アクセス権を持っているか
        if (uvo.isAdmin()) {
            return true;
        }
        return false;
    }

    // アクセス権限チェックが必要ならtrue、必要ないならfalseを返す
    public boolean isCheckRequired(ServletRequest req) {
        if (checkRequiredPath.equals(RequestUtil.getPathInfo(req))) {
            return true;
        }
        return false;
    }
}
```

AuthorizationController インタフェースを実装する。

引数 pathInfo は RequestUtil.getPathInfo(req)と同等の値が設定されている。とは限らないので注意すること。RequestUtilについては『CD-04 ユーティリティ機能』を参照のこと。

ユーザの権限情報を SampleUserValueObject の isAdmin メソッドで取得できるようにした場合。

※この実装例は簡単な説明のため、例外の発生を考慮していないことに注意すること。
※AuthorizationController 実装クラスはスレッドセーフに実装する必要があるが、この例では必要ないため特別なことをしていないことに注意すること。

◆ 拡張ポイント

アクセス権限チェック機能は、`AuthorizationController` インタフェースを実装して利用する。コーディングポイントの説明例を参照のこと。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.thin. AuthorizationControlFilter	アクセス権限を持つユーザのみ許可するように制御するフィルタクラス。
2	jp.terasoluna.fw.web.thin. AuthorizationController	<code>AuthorizationControlFilter</code> から呼び出されるクラスは、このインタフェースを実装する必要がある。
3	jp.terasoluna.fw.web.thin. UnauthorizedException	アクセス権限のないユーザがアクセスしたことを通知するための例外クラス。

■ 関連機能

- 『CD-04 ユーティリティ機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC05 アクセス権限チェック」
 - ◇ /webapps/authorization/*
 - ◇ /webapps/WEB-INF/authorization/*
 - ◇ jp.terasoluna.thin.functionsample.authorization.*
- TERASOLUNA Server Framework for Java (Web 版) チュートリアル
 - 「2.10 アクセス制御」

■ 備考

- なし

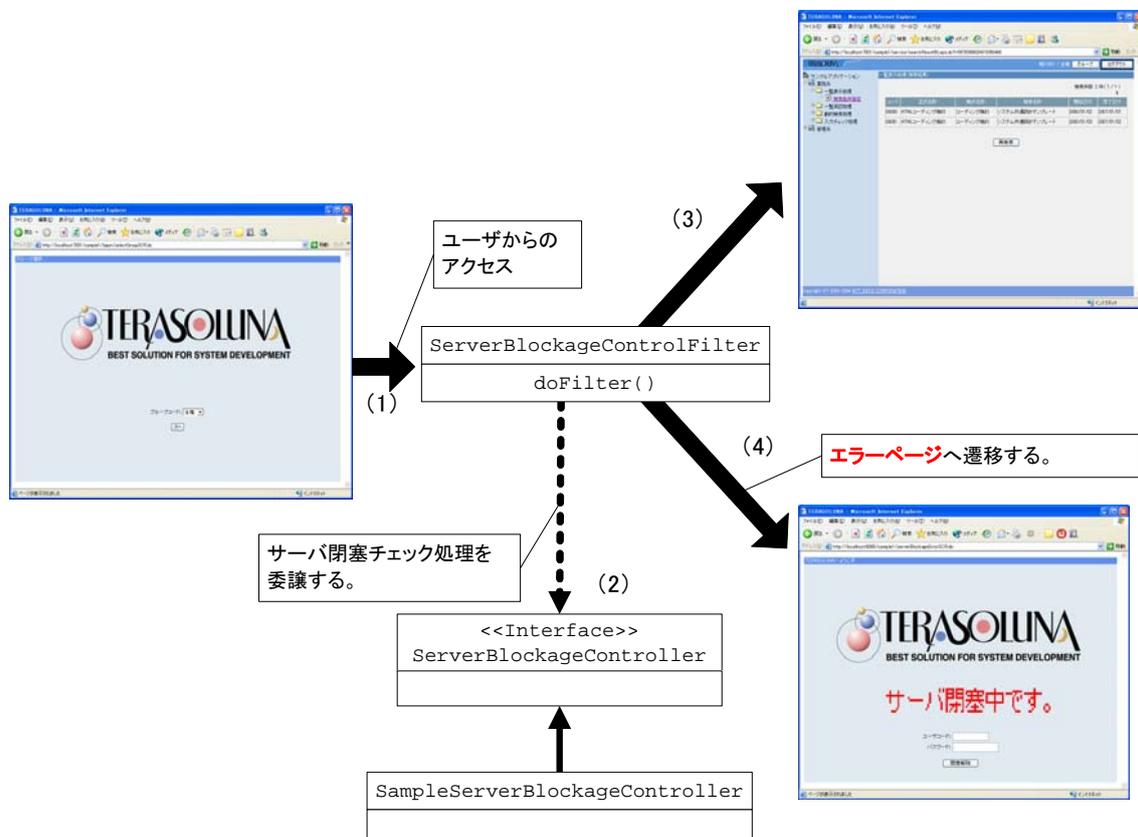
WA-03 サーバ閉塞チェック機能

■ 概要

◆ 機能概要

- ユーザからアクセスがあった時に、サーバ閉塞状態であるかをチェックする。サーバ閉塞状態であった場合は、アクセスを制限し、エラーページへ遷移する。

◆ 概念図



◆ 解説

以下の解説はデプロイメントディスクリプタ（web.xml）に `ServerBlockageControlFilter` の設定が正しくされている事が前提となる。設定方法の詳細は、コーディングポイントを参照のこと。

- (1) WebAP コンテナがユーザからのリクエストを受け `ServerBlockageControlFilter` に処理を委譲する。
- (2) `ServerBlockageControlFilter` はデプロイメントディスクリプタ (`web.xml`) の設定に従い、任意の `ServerBlockageController` インタフェース実装クラスにサーバ閉塞チェック処理を委譲する。
- (3) サーバ閉塞中ではなかった場合、次の画面に遷移する。
- (4) サーバ閉塞中であった場合、デプロイメントディスクリプタ (`web.xml`) の設定に従いエラーページへ遷移する。

■ 使用方法

◆ コーディングポイント

サーバ閉塞チェック機能は、サーブレットフィルタを用いて実現している。フィルタ内部から DI コンテナに対して、サーバ閉塞チェックの実装された `Bean` を取得するが、その際取得する `Bean` の `id` 属性についてデフォルトで用意したものを取得するか、指定したものをを使うか選択できる。

以下では `Bean` 定義ファイルにサーバ閉塞チェックを実装した `Bean` を定義する際の `id` 属性を、デフォルトで用意されているものを使う場合と、`id` 属性を指定して使う場合の 2 例について設定方法を説明する。

サーバ閉塞チェックの実装についてはどちらの場合も同じたため、最後に 1 例で説明する。

なお、『WA-01 ログオン済みチェック機能』、『WA-02 アクセス権限チェック機能』、および『WA-04 業務閉塞チェック機能』も同様の方法で実現している。

1. デフォルトの id 属性の値を使う場合の設定方法

● デプロイメントディスクリプタ (web.xml)

```

<web-app>
  .....
  <filter>
    <filter-name>serverBlockageControlFilter</filter-name>
    <filter-class>
      jp.terasoluna.fw.web.thin.ServerBlockageControlFilter
    </filter-class>
  </filter>
  .....
  <filter-mapping>
    <filter-name>serverBlockageControlFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  .....
  <error-page>
    <exception-type>
      jp.terasoluna.fw.web.thin.ServerBlockageException
    </exception-type>
    <location>/serverBlockageError.jsp</location>
  </error-page>
  .....
</web-app>

```

同名で指定する。
任意の名前でよい。

ServerBlockageControlFilter を設定する。

フィルタを動作させたい
リクエストパスを書く。

ServerBlockageException を設定する。

サーバ閉塞中だった場合に遷移させるページを設定する。

● Bean 定義ファイル

```

<beans>
  .....
  <bean id="serverBlockageController"
    class="jp.terasoluna.sample.SampleServerBlockageController"/>
  .....
</beans>

```

デフォルトの id 属性の値
serverBlockageControlFilter を設定する。

ServerBlockageController の
実装クラスを設定する。

2. id 属性の値を変えて使う場合の設定方法

● デプロイメントディスクリプタ (web.xml)

```
<web-app>
.....
<filter>
  <filter-name>serverBlockageControlFilter</filter-name>
  <filter-class>
    jp.terasoluna.fw.web.thin.ServerBlockageControlFilter
  </filter-class>
  <init-param>
    <param-name>controller</param-name>
    <param-value>
      sampleServerBlockageController
    </param-value>
  </init-param>
</filter>
.....
<filter-mapping>
  <filter-name>serverBlockageControlFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
.....
<error-page>
  <exception-type>
    jp.terasoluna.fw.web.thin.ServerBlockageException
  </exception-type>
  <location>/serverBlockageError.jsp</location>
</error-page>
.....
</web-app>
```

ServerBlockageControlFilterを設定する。

パラメータ名は controller と設定する。

同名で指定する。任意の名前でよい。

フィルタを動作させたいリクエストパスを書く。

ServerBlockageExceptionを設定する。

同名で指定する。任意の名前でよい。

サーバ閉塞中だった場合に遷移させるページを設定する。

● bean 定義ファイル

```
<beans>
.....
<bean id="sampleServerBlockageController"
  class="jp.terasoluna.sample.SampleServerBlockageController"/>
.....
</beans>
```

ServerBlockageController の実装クラスを設定する。

3. ServerBlockageController の実装クラス例

```
package jp.terasoluna.sample;

import java.util.Date;
import javax.servlet.ServletException;
import jp.terasoluna.fw.web.thin.ServerBlockageController;

public class SampleServerBlockageController
    implements ServerBlockageController {

    // 開放状態
    public static final int OPEN = 0;
    // 予閉塞状態
    public static final int PRE_BLOCKADED = 1;
    // 閉塞状態
    public static final int BLOCKADED = 2;

    // サーバ閉塞していても通すパス
    private String[] alwaysOpenPaths
        = {"/admin/index.jsp", "/admin/action.do"};

    // サーバ閉塞状態かどうかを表すフラグ
    private int state = OPEN;

    // 閉塞状態に遷移する時間
    private Date blockadingDate = null;
}
```

ServerBlockageController インタフェースを実装する。

※続く

```
// サーバ閉塞状態の場合はtrueを返し、そうでない場合はfalseを返す
public boolean isBlockaded(String pathInfo) {
    // サーバ閉塞状態であってもチェックしないパスである場合、サーバ閉塞状態でないとする
    for (int i = 0; i < alwaysOpenPaths.length; ++i) {
        if (alwaysOpenPaths[i].equals(pathInfo)) {
            return false;
        }
    }
    // サーバの状態を確認する
    if (state == BLOCKADED) {
        return true;
    }
    // 閉塞遷移時間が指定されてない場合は閉塞しない
    if (blockadingDate == null) {
        return false;
    }
    // 閉塞遷移時間が過ぎていたら閉塞する
    synchronized (this) {
        if (blockadingDate != null) {
            Date now = new Date();
            // 閉塞状態に遷移する時間がすでに過ぎていたら閉塞状態に遷移する
            if (blockadingDate.before(now)) {
                state = BLOCKADED;
            }
        }
    }
    if (state == BLOCKADED) {
        return true;
    }
    return false;
}
```

jp.co.nttdata.terasoluna.fw.web.RequestUtil.getPathInfo(req)の形式で渡される。

例ではサーバの状態をこのインスタンスが管理しているため、不整合が起こらないようロックしている。

※続く

```
public boolean isBlocked() {
    // パス情報を使わないため、オーバーロードされたメソッドに委譲する。
    return isBlocked(null);
}

// 予閉塞状態か判定する。
public boolean isPreBlocked() {
    if (state == BLOCKADED || state == PRE_BLOCKADED) {
        return true;
    }
    return false;
}
```

以降のメソッドについてはフィルタからは呼ばれないが、サーバ閉塞に必要なメソッドである。一般的に、管理画面等から呼び出され、閉塞・開放を行う。

// サーバを閉塞する

```
public void blockade() {
    synchronized (this) {
        state = BLOCKADED;
    }
}
```

}

// サーバを開放する

```
public void open() {
    synchronized (this) {
        state = OPEN;
        blockadingDate = null;
    }
}
```

}

// サーバを予閉塞状態にする

```
public void preBlockade() {
    synchronized (this) {
        state = PRE_BLOCKADED;
    }
}
```

}

// サーバを予閉塞状態にし、指定された時刻に閉塞する

```
public void preBlockade(Date time) {
    synchronized (this) {
        state = PRE_BLOCKADED;
        blockadingDate = time;
    }
}
```

}

例ではサーバの状態をこのインスタンスが管理しているため、不整合が起らないようロックしている。

※この実装例は簡単な説明のため、例外の発生を考慮していないことに注意すること。

◆ 拡張ポイント

- サーバ閉塞チェック機能は、**ServerBlockageController** インタフェースを実装して利用する。コーディングポイントの説明例を参照のこと。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.thin.ServerBlockageControlFilter	サーバ閉塞状態の場合、アクセスを許可しないように制御するフィルタクラスである。
2	jp.terasoluna.fw.web.thin.ServerBlockageController	ServerBlockageControlFilter から呼び出されるクラスは、このインタフェースを実装する必要がある。
3	jp.terasoluna.fw.web.thin.ServerBlockageException	サーバ閉塞状態の場合にアクセスしたことを通知するための例外クラスである。

■ 関連機能

- なし

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC06 サーバ閉塞チェック」
 - ◇ /webapps/serverblockage/*
 - ◇ /webapps/WEB-INF/serverblockage/*
 - ◇ jp.terasoluna.thin.functionsample.serverblockage.*

■ 備考

- なし

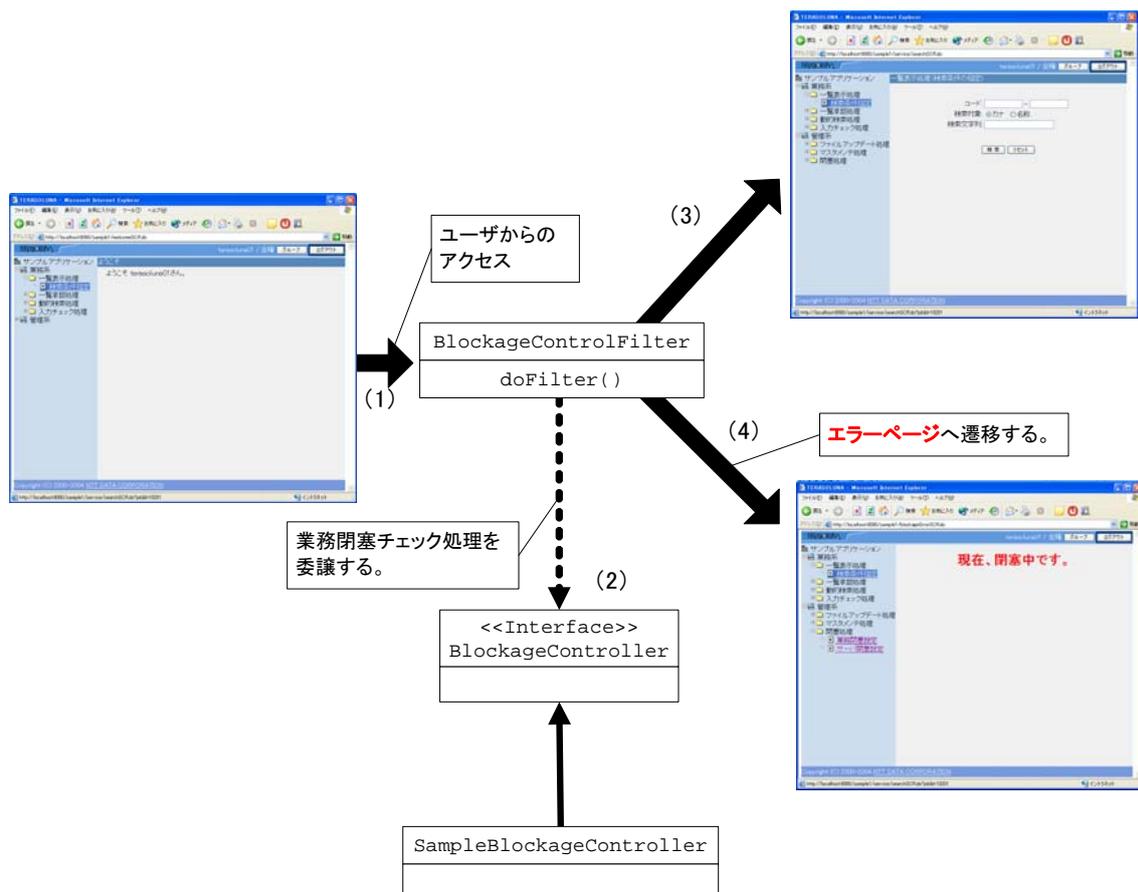
WA-04 業務閉塞チェック機能

■ 概要

◆ 機能概要

- ユーザからアクセスがあった時に、業務閉塞状態であるかをチェックする。業務閉塞状態であった場合は、アクセスを制限し、エラーページへ遷移する。
- ※ この機能を利用する場合には、リクエストパスから業務が判別できるように設計しなければならない。

◆ 概念図



◆ 解説

以下の解説はデプロイメントディスクリプタ（web.xml）に `BlockageControlFilter` の設定が正しくされている事が前提となる。設定方法の詳細は、コーディングポイントを参照のこと。

- (1) WebAP コンテナがユーザからのリクエストを受け `BlockageControlFilter` に処理を委譲する。
- (2) `BlockageControlFilter` はデプロイメントディスクリプタ（web.xml）の設定に従い、任意の `BlockageController` インタフェース実装クラスに業務閉塞チェック処理を委譲する。
- (3) 業務閉塞中ではなかった場合、次の画面に遷移する。
- (4) 業務閉塞中であった場合、デプロイメントディスクリプタ（web.xml）の設定に従いエラーページへ遷移する。

■ 使用方法

◆ コーディングポイント

業務閉塞チェック機能は、サーブレットフィルタを用いて実現している。フィルタ内部から DI コンテナに対して、業務閉塞チェックの実装された `Bean` を取得するが、その際取得する `Bean` の `id` 属性についてデフォルトで用意したものを取得するか、指定したものをを使うか選択できる。

以下では `Bean` 定義ファイルに業務閉塞チェックを実装した `Bean` を定義する際の `id` 属性を、デフォルトで用意されているものを使う場合と、`id` 属性を指定して使う場合の 2 例について設定方法を説明する。

業務閉塞チェックの実装についてはどちらの場合も同じたため、最後に 1 例で説明する。

なお、『WA-01 ログオン済みチェック機能』、『WA-02 アクセス権限チェック機能』、および『WA-03 サーバー閉塞チェック機能』も同様の方法で実現している。

1. デフォルトの id 属性の値を使う場合の設定方法

● デプロイメントディスクリプタ (web.xml)

```

<web-app>
.....
<filter>
  <filter-name>blockageControlFilter</filter-name>
  <filter-class>
    jp.terasoluna.fw.web.thin.BlockageControlFilter
  </filter-class>
</filter>
.....
<filter-mapping>
  <filter-name>blockageControlFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
.....
<error-page>
  <exception-type>
    jp.terasoluna.fw.web.thin.BlockageException
  </exception-type>
  <location>/blockageError.jsp</location>
</error-page>
.....
</web-app>

```

同名で指定する。
任意の名前でよい。

BlockageControlFilter を設定する。

フィルタを動作させたい
リクエストパスを書く。

BlockageException を設定する。

業務閉塞中だった場合に遷移させるページを設定する。

● Bean 定義ファイル

```

<beans>
.....
  <bean id="blockageController"
class="jp.terasoluna.sample.SampleBlockageController" />
.....
</beans>

```

デフォルトの id 属性の値
blockageController を指定する。

BlockageController の実装クラスを設定する。

2. id 属性の値を変えて使う場合の設定方法

● デプロイメントディスクリプタ (web.xml)

```
<web-app>
.....
<filter>
  <filter-name>blockageControlFilter</filter-name>
  <filter-class>
    jp.terasoluna.fw.web.thin.BlockageControlFilter
  </filter-class>
  <init-param>
    <param-name>controller</param-name>
    <param-value>
      sampleBlockageController
    </param-value>
  </init-param>
</filter>
.....
<filter-mapping>
  <filter-name>blockageControlFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
.....
<error-page>
  <exception-type>
    jp.terasoluna.fw.web.thin.BlockageException
  </exception-type>
  <location>/blockageError.jsp</location>
</error-page>
.....
</web-app>
```

BlockageControlFilter を設定する。

パラメータ名は controller と設定する。

同名で指定する。
任意の名前でよい。

フィルタを動作させたい
リクエストパスを書く。

BlockageException を
設定する。

同名で指定する。
任意の名前でよい。

業務閉塞中だった場合に遷移させる
ページを設定する。

● bean 定義ファイル

```
<beans>
.....
<bean id="sampleBlockageController"
  class="jp.terasoluna.sample.SampleBlockageController"/>
.....
</beans>
```

BlockageController の実装クラス
を設定する。

3. BlockageController の実装クラス例

```
package jp.terasoluna.sample;

import javax.servlet.ServletException;
import jp.terasoluna.fw.web.thin.BlockageController;

public class SampleBlockageController
    implements BlockageController {

    // チェックしないパス
    private String noCheckPath = "/sample/index.jsp";

    // 業務閉塞しているパス
    private HashSet<String> blockadedPaths = new HashSet<String>();

    // 業務閉塞状態の場合はtrueを返し、そうでない場合はfalseを返す
    public boolean isBlockaded(String path,
        ServletRequest req) {
        synchronized (blockadedPaths) {
            if (blockadedPaths.contains(path)) {
                return true;
            }
            return false;
        }
    }

    // 業務閉塞チェックが必要かどうか判定し、必要ならtrue, 必要ないならfalseを返す
    public boolean isCheckRequired(ServletRequest request) {
        // チェックする必要がないパスへのリクエストであるか判定する
        if (noCheckPath.equals(RequestUtil.getPathInfo(req))) {
            return false;
        }
        return true;
    }
}
```

BlockageController インタフェースを実装する。

RequestUtil.getPathInfo(req)と同等とは限らないことに注意すること。

引数 req はユーザを識別したい場合に利用する。

スレッドセーフな実装にする。
blockadedPaths フィールドが他のスレッドによって状態を変更されないように同期化している。

※続く

以降のメソッドについてはフィルタからは呼ばれないが、業務閉塞に必要なメソッドである。

```
public boolean isBlockaded(String path) {  
    // リクエスト情報を使わないため、オーバーロードされたメソッドに委譲する  
    isBockaded(path, null);  
}
```

```
public void blockade(String path) {  
    // リクエスト情報を使わないため、オーバーロードされたメソッドに委譲する  
    blockade(path, null);  
}
```

引数 path は、業務閉塞フィルタがパス単位で閉塞することを想定しているため、閉塞するパスが渡される。

```
// 指定された、業務を呼び出すパスを閉塞する  
public void blockade(String path,  
                    ServletRequest req) {  
    synchronized (blockadedPaths) {  
        blockadedPaths.add(path);  
    }  
}
```

引数 req はユーザを識別したい場合に利用する。

スレッドセーフな実装にする。
blockadedPaths フィールドが他のスレッドによって状態を変更されないように同期化している。

```
public void open(String path) {  
    // リクエスト情報を使わないため、オーバーロードされたメソッドに委譲する  
    open(path, null);  
}
```

```
// 指定された、業務を呼び出すパスを開放する  
public void open(String path,  
                ServletRequest req) {  
    synchronized (blockadedPaths) {  
        blockadedPaths.remove(path);  
    }  
}
```

スレッドセーフな実装にする。
blockadedPaths フィールドが他のスレッドによって状態を変更されないように同期化している。

※この実装例は簡単な説明のため、例外の発生を考慮していないことに注意すること。

◆ 拡張ポイント

- 業務閉塞チェック機能は、**BlockageController** インタフェースを実装して利用する。コーディングポイントの説明例を参照のこと。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.thin.B lockageControlFilter	業務閉塞状態の場合、閉塞している業務へのアクセスを許可しないように制御するフィルタクラスである。
2	jp.terasoluna.fw.web.thin.B lockageController	BlockageControlFilter から呼び出されるクラスは、このインタフェースを実装する必要がある。
3	jp.terasoluna.fw.web.thin.B lockageException	業務閉塞状態の業務にアクセスしたことを通知するための例外クラスである。

■ 関連機能

- なし。

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC07 業務閉塞チェック」
 - ◇ /webapps/blockage/*
 - ◇ /webapps/WEB-INF/blockage/*
 - ◇ jp.terasoluna.thin.functionsample.blockage.*

■ 備考

- なし

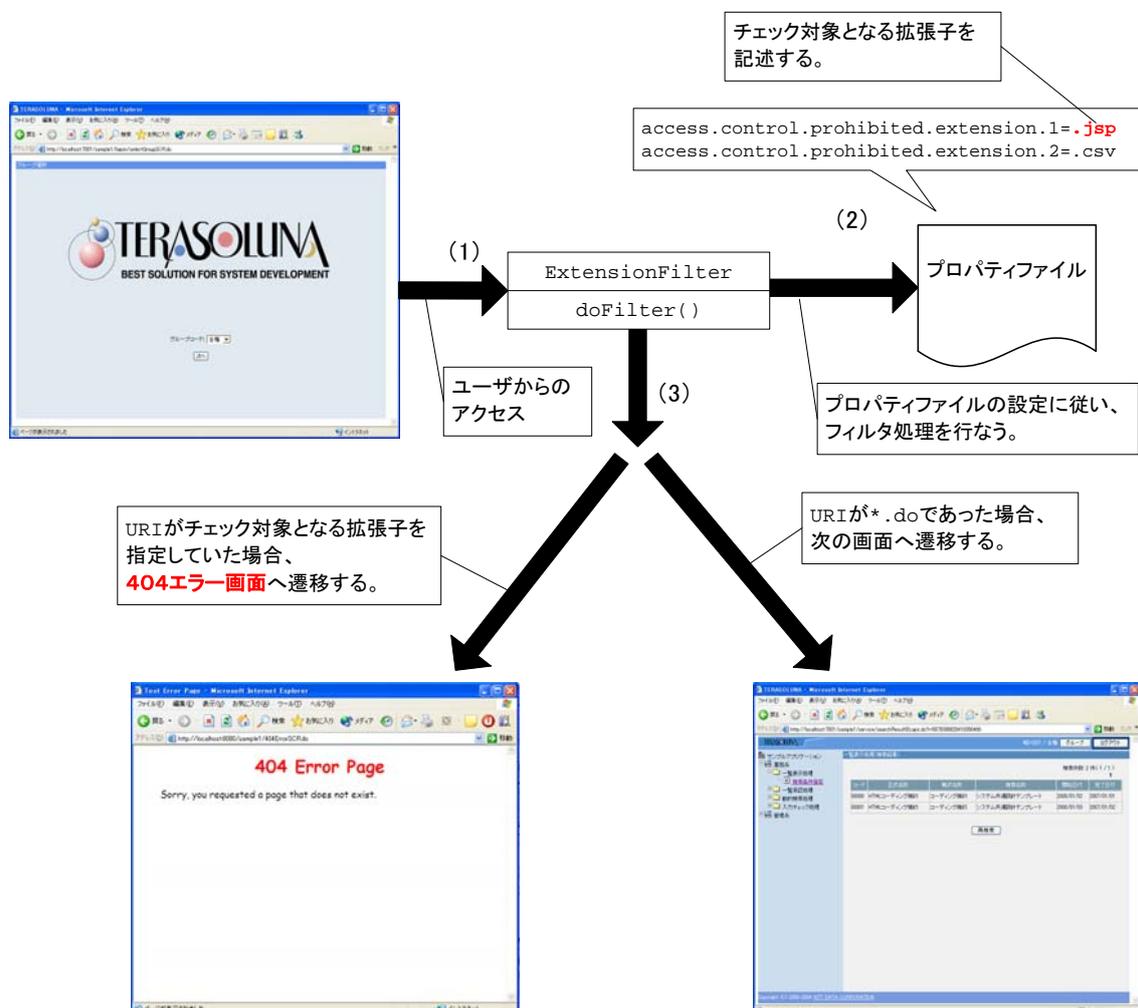
WA-05 拡張子直接アクセス禁止機能

■ 概要

◆ 機能概要

- ブラウザからのリクエストに対し、任意の拡張子をもつコンテンツへのアクセスを制限する。

◆ 概念図



◆ 解説

以下の解説はデプロイメントディスクリプタ（web.xml）に `ExtensionFilter` の設定が正しくされている事が前提となる。設定方法の詳細は、コーディングポイントを参照のこと。

- (1) WebAP コンテナがユーザからのリクエストを受け、`ExtensionFilter` に処理を委譲する。
- (2) プロパティファイルの設定に従って禁止された拡張子かどうかを判定する。
- (3) 禁止されている拡張子を指定されていた場合は、404 エラー画面へ遷移する。

■ 使用方法

◆ コーディングポイント

このフィルタを利用するにあたって、デプロイメントディスクリプタ（web.xml）にフィルタとしての設定をするほか、直接アクセスを禁止したい拡張子のリストをシステム設定プロパティファイル（`system.properties`）に書く。また、禁止するようにした拡張子をもつコンテンツであっても、例外として通過させたいものについては、同様にシステム設定プロパティファイル（`system.properties`）に書く。この場合は拡張子ではなく、個別にパスを書くことになる。書き方については以下の例を参考のこと。

なおシステム設定プロパティファイル（`system.properties`）の扱いについては 『CD-01 ユーティリティ機能』を参照のこと。

● デプロイメントディスクリプタ (web.xml)

```

<filter>
  <filter-name>extensionFilter</filter-name>
  <filter-class>
    jp.terasoluna.fw.web.thin.ExtensionFilter
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>extensionFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

```

同名で指定する。
任意の名前でよい。

ExtensionFilter を設定する。

拡張子直接アクセス禁止フィルタを動作させたい
リクエストパスのパターンを指定する。

● システム設定プロパティファイル (system.properties)

```

#-----
# ExtensionFilterの拡張子ごとの直接アクセス制限チェック対象の拡張子を
# 1から順に指定する。
access.control.prohibited.extension.1=.jsp
access.control.prohibited.extension.2=.htm
access.control.prohibited.extension.3=.html
access.control.prohibited.extension.4=.css

#-----
# ExtensionFilterの拡張子制限チェックを
# 対象外にするパスを1から順に指定する。
restrictionEscape.1=/index.jsp
restrictionEscape.2=/common.css

```

1 から始まる連続した番号を指定する。

制限する拡張子のプロパティキーは
access.control.prohibited.extension.+ <通番>

1 から始まる連続した番号を指定する。

通過させるパスは**コンテキスト相対パス**で書く。

例外として通過させるパスのプロパティキーは
restrictionEscape.+<通番>

※<通番>となっている部分は必ず「1」から順に連続した番号を振ること。
途中の番号が欠けている場合、欠けた番号以降は無効となる。

◆ 拡張ポイント

なし。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.thin.ExtensionFilter	指定された拡張子への直接アクセスを禁止するフィルタクラス。

■ 関連機能

- 『CD-01 ユーティリティ機能』

■ 使用例

- Terasoluna Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC08 拡張子直接アクセス禁止」
 - ◇ /webapps/extension/*
 - ◇ /webapps/WEB-INF/extension/*
 - ◇ jp.terasoluna.thin.functionsample.extension.*

■ 備考

- なし。

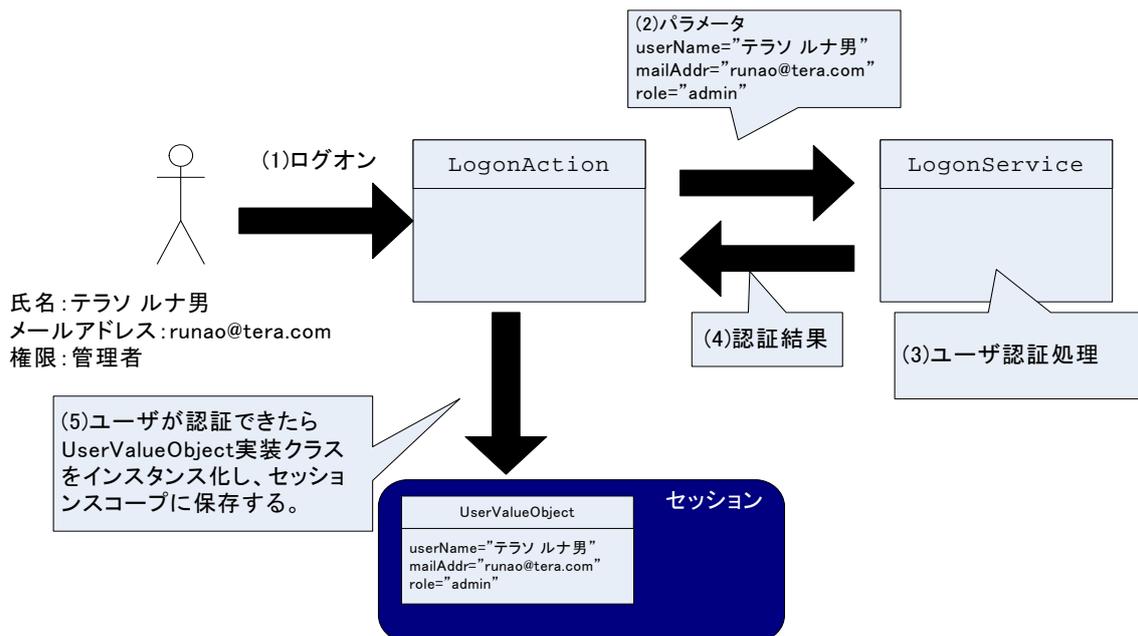
WB-01 ユーザ情報保持機能

■ 概要

◆ 機能概要

- ログオン中のユーザ情報を保持し、セッションに格納されるバリューオブジェクトを提供する。
- ログオン中のユーザ情報をビジネスロジックに引き渡し、ビジネスロジック中でユーザ情報を利用できるようにする。
- ユーザバリューオブジェクト (UVO) はシステム固有のものなので簡単に入れ替えられるようにする。

◆ 概念図



◆ 解説

- (1) まだ、アプリケーションによって認証されていないユーザがログイン画面からアクセスする。
- (2) ログイン認証処理を制御するアクションクラス（任意）は必要なパラメータをビジネスロジック（任意）に渡し、認証処理を委譲する。
- (3) ビジネスロジックはアプリケーションの仕様からアクセスしたユーザの認証チェック処理を実行する。
- (4) ビジネスロジックは認証処理の結果をアクションクラスに返却する。
- (5) アクションクラスはビジネスロジックの結果を受け取り、ユーザが認証された場合は `UserValueObject` 実装クラスのインスタンスを作成し、必要な情報を設定して、セッションに保存する。

■ 使用方法

◆ コーディングポイント

ユーザ情報保持クラスは TERASOLUNA Server Framework for Java (Web 版)から提供された抽象クラスを継承して作成する。実装クラス名をプロパティファイルに記述することによって、ユーティリティメソッドを使用してユーザ情報保持クラスのインスタンス作成が可能である。

- プロパティファイル

```
user.value.object=jp.terasoluna.sample.xxxx.SampleUVO
```

user.value.object をキーとして、UserValueObject 実装クラス名を設定する。

- UserValueObject 実装クラス

```
public class SampleUVO extends UserValueObject {
    /**ユーザ名*/
    private String userName = null;
    /**ユーザロール*/
    private String userRole = null;
    /**ユーザ名を設定する。*/
    public void setUsername(String userName) {
        this.userName = userName;
    }
    /**ユーザ名を取得する*/
    public String getUsername() {
        return this.userName
    }
    .....
}
```

必要な属性とアクセサメソッドを定義する。

- ユーザ認証を実行するアクションクラス

```
public class AuthenticationAction
    extends AbstractBLogicAction<AuthenticationParams> {
    public BLogicResult doExecuteBLogic(AuthenticationParams params)
        throws Exception {
        //ビジネスロジック呼び出し
        if (authenticated) {
            SampleUVO uvo = (SampleUVO) UserValueObject.createUserValueObject();
            BLogicResult result = new BLogicResult();
            result.setResultObject(uvo);
            .....
        }
    }
}
```

※ ビジネスロジックの実装、ビジネスロジックの結果のセッションへの反映については『WH-01 ビジネスロジック実行機能』、『WH-02 ビジネスロジック入出力機能』を参照のこと。

◆ 拡張ポイント

なし。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.User ValueObject	ユーザ情報保持クラスが継承する抽象クラス。プロパティファイルに記述したクラス名から実装クラスインスタンスを生成する。

■ 関連機能

- 『WA-01 ログオン済みチェック機能』
- 『WA-02 アクセス権限チェック機能』
- 『WH-01 ビジネスロジック実行機能』
- 『WH-02 ビジネスロジック入出力機能』

■ 使用例

- Terasoluna Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC09 ユーザ情報保持」
 - ◇ /webapps/uvo/*
 - ◇ /webapps/WEB-INF/uvo/*
 - ◇ jp.terasoluna.thin.functionsample.common.FunctionUVO.java
- Terasoluna Server Framework for Java (Web 版) チュートリアル
 - 「2.4 ログオン」
 - ログイン画面

■ 備考

なし。

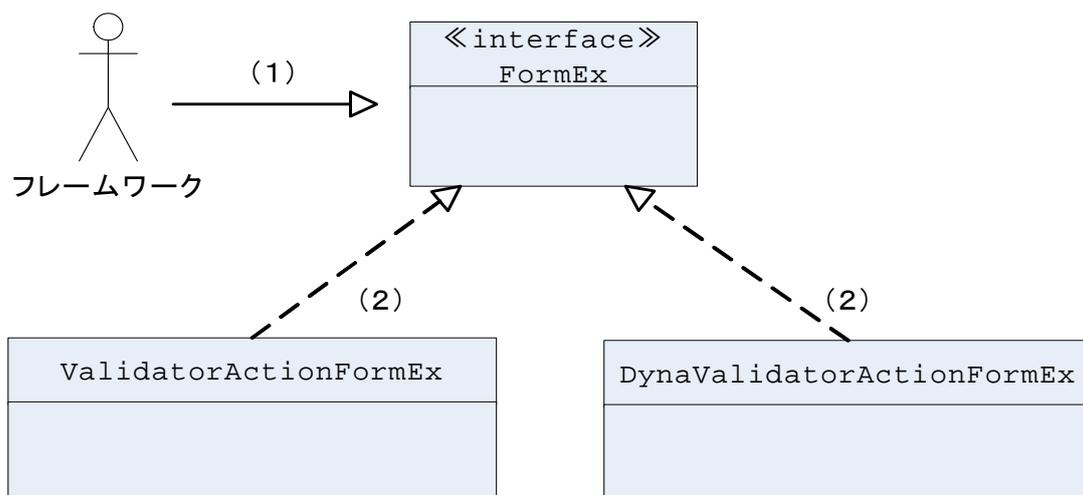
WB-02 アクションフォーム拡張機能

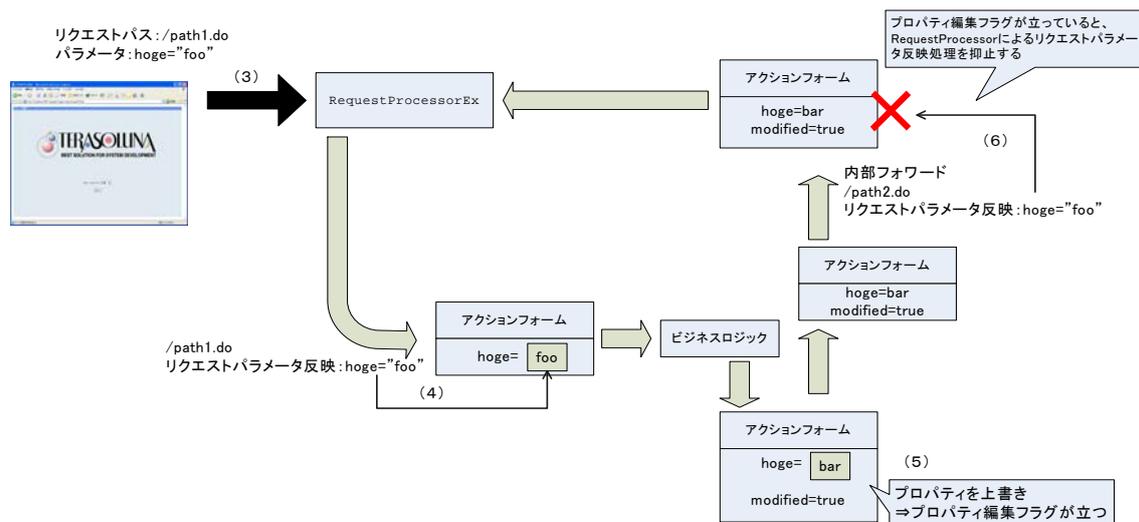
■ 概要

◆ 機能概要

- TERASOLUNA Server Framework for Java (Web 版)内部で動的アクションフォームと静的アクションフォームを意識せず操作できるように共通のインタフェース、およびインタフェースを実装した基底クラスを提供する。
- プロパティの典型的な操作をサポートするメソッドを提供する。
- ビジネスロジックを実行し、アクションフォームのプロパティ値が変更された場合、その後の内部フォワード時にリクエストパラメータの上書きを抑止する機能を提供する。
- "_"が先頭に付いたアクションフォームをセッション中に常に一つしか存在しないことを保証する機能を提供する。
『WB-03 アクションフォーム切替機能』を参照のこと。
- 設定ファイルをベースとしたプロパティの自動初期化機能を提供する。
『WB-04 フォームプロパティリセット機能』を参照のこと。
- commons-validator のルールを拡張し、典型的な入力チェックルールを提供する。
『WF-01 入力チェック拡張機能』を参照のこと。

◆ 概念図





◆ 解説

TERASOLUNA Server Framework for Java (Web 版)からのアクションフォームの操作

- (1) TERASOLUNA Server Framework for Java (Web 版)はアクションフォームの共通インタフェースである FormEx にアクセスすることで、実装するクラスが動的なアクションフォームであるか静的なアクションフォームであることを意識することなくアクションフォームを操作することが可能である。
- (2) FormEx を実装するクラスとして、動的なアクションフォームである DynaValidatorActionFormEx と静的なアクションフォームである ValidatorActionFormEx を提供する。

リクエストパラメータ上書き抑止機能

- ※ この機能は Struts 設定ファイルに RequestProcessorEx の設定が正しく行われていることを前提とする。
- (3) ユーザが画面からサーバ側へリクエストを送信する。
 - (4) RequestProcessor の processPopulate メソッドによりリクエストパラメータが該当アクションフォームに反映される。
 - (5) ビジネスロジックを呼び出し、(4)で反映されたプロパティの値を変更すると、アクションフォーム内にプロパティが編集されたことを示すフラグが立つ。
 - (6) ビジネスロジックの終了後内部フォワード処理を行うと、RequestProcessor の processPopulate が再度呼び出され、リクエストパラメータの反映を行うため、(5)で変更したプロパティをリクエストパラメータで上書きしようとするが、プロパティ編集フラグが立っているため、processPopulate メソッドの処理をキャンセルする。

■ 使用方法

◆ コーディングポイント

DynaValidatorActionFormEx は Struts 設定ファイルにフォーム名、プロパティ定義を設定する。

- Struts 設定ファイル (struts-config.xml)

```
<form-bean name="logonForm"
  type="jp.terasoluna.fw.web.struts.form.DynaValidatorActionFormEx" >
  <form-property name="userId" type="java.lang.String" />
</form-bean>
```

● 設定方法は通常の DynaActionForm と同様

ValidatorActionFormEx を使用する場合は継承して実装クラスを作成する。

- ValidatorActionFormEx 実装クラスの例

```
public class SampleForm extends ValidatorActionFormEx {
    private String userName = null;
    public void setUserName(String userName) {
        this.userName = userName;
    }
    public String getUserName() {
        return this.userName;
    }
}
```

DynaValidatorActionFormEx、ValidatorActionFormEx の配列/List プロパティ操作メソッドではインデックス範囲外の要素にアクセスした場合に例外を発生させないように実装されている。

- 配列/List 操作メソッド

```
DynaValidatorActionFormEx formEx = (FormEx) form;
String[] stringArray = new String[] {
    "a", "b", "c"
};
formEx.set("hoge", stringArray); //hogeプロパティに要素数3の配列を設定
formEx.get("hoge", 5); //配列の要素範囲外なので通常なら例外が発生するが、
//nullが返却される。
formEx.set("hoge", 3, "c"); //配列の要素範囲外なので通常なら例外が発生するが
//配列の要素数を4に増やし、4番目の要素に"c"
//を代入する。
```

◆ 拡張ポイント

なし。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.struts.form.FormEx	TERASOLUNA Server Framework for Java (Web 版)がアクセスするアクションフォームの共通インタフェース。
2	jp.terasoluna.fw.web.struts.form.DynaValidatorActionFormEx	クラスの実装が不要な動的な FormEx 実装クラス。
3	jp.terasoluna.fw.web.struts.form.ValidatorActionFormEx	静的な FormEx 実装クラス。このクラスを継承してアクションフォームを実装する。
4	jp.terasoluna.fw.web.struts.form.ActionFormUtil	FormEx の操作に関わるユーティリティメソッドを提供するクラス。

■ 関連機能

- 『WB-03 アクションフォーム切替機能』
- 『WB-04 フォームプロパティリセット機能』
- 『WF-01 入力チェック拡張機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC10 アクションフォーム拡張」
 - ◇ /webapps/formex/*
 - ◇ /webapps/WEB-INF/formex/*
 - ◇ jp.terasoluna.thin.functionsample.formex.*
- TERASOLUNA Server Framework for Java (Web 版) チュートリアル
 - 「2.4 ログオン」
 - 「2.5 一覧表示」
 - /webapps/WEB-INF/struts-config.xml

■ 備考

なし。

WB-03 アクションフォーム切替機能

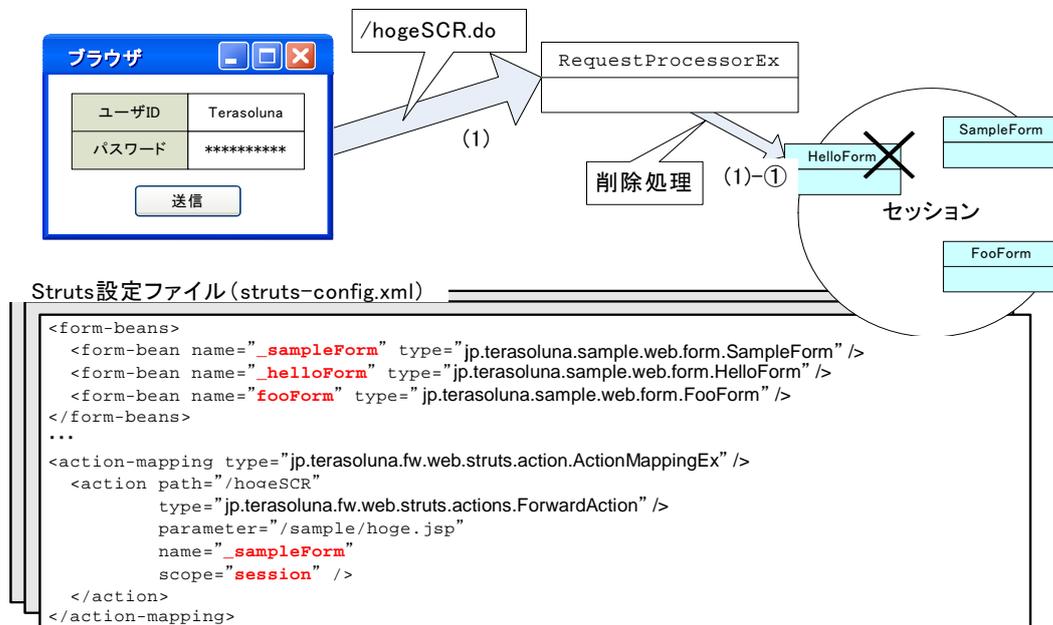
■ 概要

◆ 機能概要

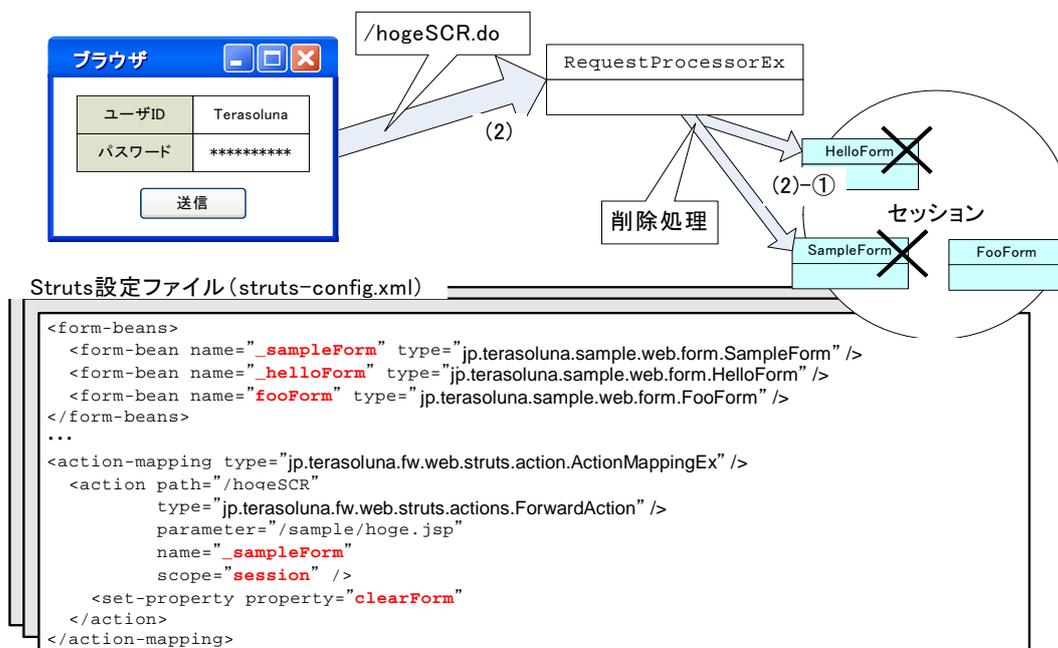
- Terasoluna Server Framework for Java (Web 版)が提供する RequestProcessorEx を使用し、アクションフォーム名の先頭に "_" を付与することで、 "_" 付のアクションフォームは常にセッション上に一つしか存在しないことを保証する機能。この機能を利用し、ユースケース毎にアクションフォームを定義することで、同一ユースケース内の操作であれば同じアクションフォームを使用し、別のユースケースに遷移したら、前のユースケースで使用していたアクションフォームを自動的に削除することが可能である。
- 複数画面を同時に立ち上げ、1 ユーザが同時に複数のユースケースを操作するようなアプリケーションでは本機能を使用することはできない。

◆ 概念図

“_”から始まる論理名をもつActionFormに紐づいたパスにアクセスすると・・・



“_”から始まる論理名をもつActionFormに紐づき、かつ“clearForm”プロパティに“true”を設定したパスにアクセスすると・・・



◆ 解説

- (1) クライアントから“_”から始まる論理名を持ち、かつセッションスコープに格納される ActionForm に紐づいたパスに対してリクエストが発生した場合。
 - (1)-① RequestProcessorEx により対象となるアクションパスに紐づいた ActionForm 以外の“_”から始まる論理名を持つ ActionForm をセッションから削除する。
- (2) クライアントから“_”から始まる論理名を持ち、かつセッションスコープに格納される ActionForm に紐づき、さらに clearForm プロパティに“true”が設定されているパスに対してリクエストが発生した場合。
 - (2)-① RequestProcessorEx により“_”から始まる論理名を持つ全ての ActionForm をセッションから削除する。

■ 使用方法

◆ コーディングポイント

RequestProcessorEx の設定を行う。アクションフォーム名の先頭に“_”を付与する。

- Struts 設定ファイル

```
<form-beans>
  <form-bean name="_sampleForm"
             type="jp.terasoluna.sample.web.form.SampleForm" />
</form-beans>
<controller
  processorClass="jp.terasoluna.fw.web.struts.action.RequestProcessorEx"/>
```

フォーム名の先頭は“_”

アクションフォームは session スコープで使用する。

- Struts 設定ファイル

```
<action-mappings
  type=" jp.terasoluna.fw.web.struts.action.ActionMappingEx">
  <action
    path="/hogeSCR"
    type=" jp.terasoluna.fw.web.struts.actions.ForwardAction"
    parameter="/sample/hoge.jsp"
    name="_sampleForm"
    scope="session" />
</action-mappings>
```

clearForm プロパティを設定することでセッション上に存在する全ての "_"付アクションフォームを削除することができる。

- Struts 設定ファイル

```
<action-mappings
  type=" jp.terasoluna.fw.web.struts.action.ActionMappingEx">
  <action
    path="/hogeSCR"
    type=" jp.terasoluna.fw.web.struts.actions.ForwardAction"
    parameter="/sample/hoge.jsp"
    name="_sampleForm"
    scope="session" />
    <set-property property="clearForm" value="true" />
  </action-mappings>
```

◆ 拡張ポイント

なし。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.struts.action.RequestProcessorEx	アクションフォームの切替処理を行う。

■ 関連機能

- 『WB-02 アクションフォーム拡張機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC11 アクションフォーム切り替え」
 - ◇ /webapps/formtrans/*
 - ◇ /webapps/WEB-INF/formtrans/*

■ 備考

なし。

WB-04 フォームプロパティリセット機能

■ 概要

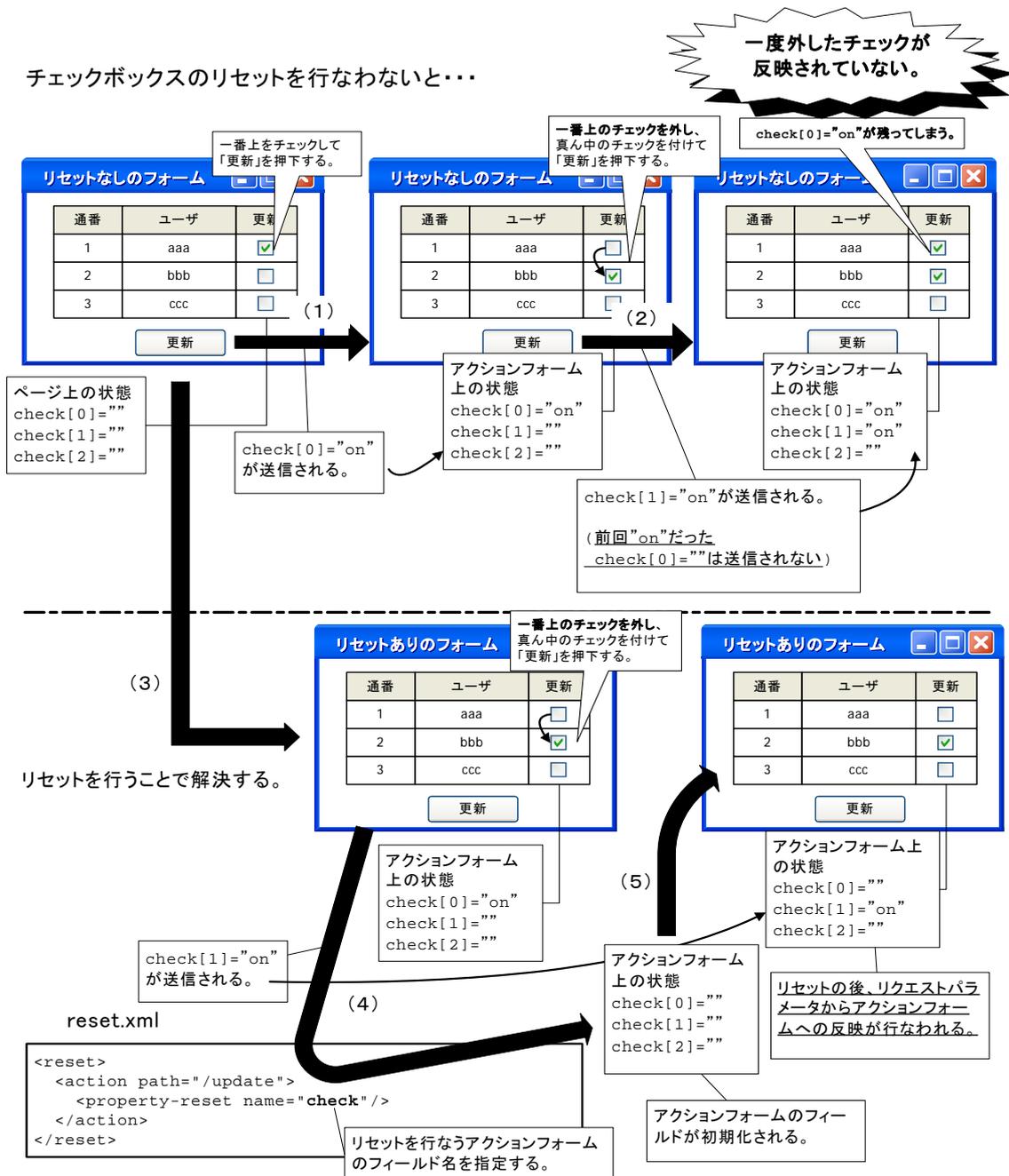
◆ 機能概要

- Struts ではユーザにより送信されたリクエストパラメータをアクションフォームに反映する前に、アクションフォームに対し、**reset** メソッドを実行する。本機能はアクションフォームの **reset** メソッドをオーバーライドし、**Resetter** インタフェースを実装したクラスにリセット処理を委譲する仕組みを提供する。
- TERASOLUNA Server Framework for Java (Web 版)は **Resetter** インタフェースのデフォルト実装クラスをである **ResetterImpl** クラスを提供する。このクラスは設定ファイルを元にアクションフォームの指定フィールドをリクエストパラメータ反映前に自動的に初期化する処理を提供する。
- **ResetterImpl** クラスは複数ページに渡る一覧表示中にチェックボックスが存在する場合は、現在表示しているページに該当するチェックボックスのプロパティ値のみを初期化する必要があるが、配列や **List** 型のプロパティの一定の範囲のみを初期化する機能を提供する。
- **reset** メソッドはアクションフォームを **session** スコープで使用した場合に、チェックボックス、セレクトボックス、ラジオボタンなどのフィールド値を格納するプロパティの初期化を目的に提供されている。その他のフィールドの初期化は本機能を使用せず、ビジネスロジックなどを経由してプロパティ値の初期化を行うこと。

◆ 概念図

下記は `ResetterImpl` を使用した場合の HTML フォームのチェックボックスのリセットの例である。

チェックボックスのリセットを行なわないと・・・



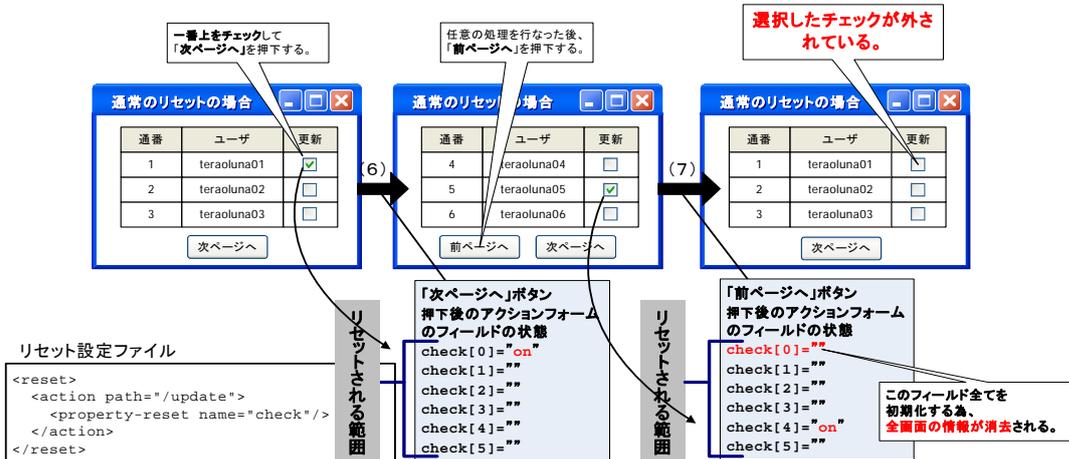
◆ 解説

以下、操作対象のアクションフォームがセッションに格納されていることを前提とする。

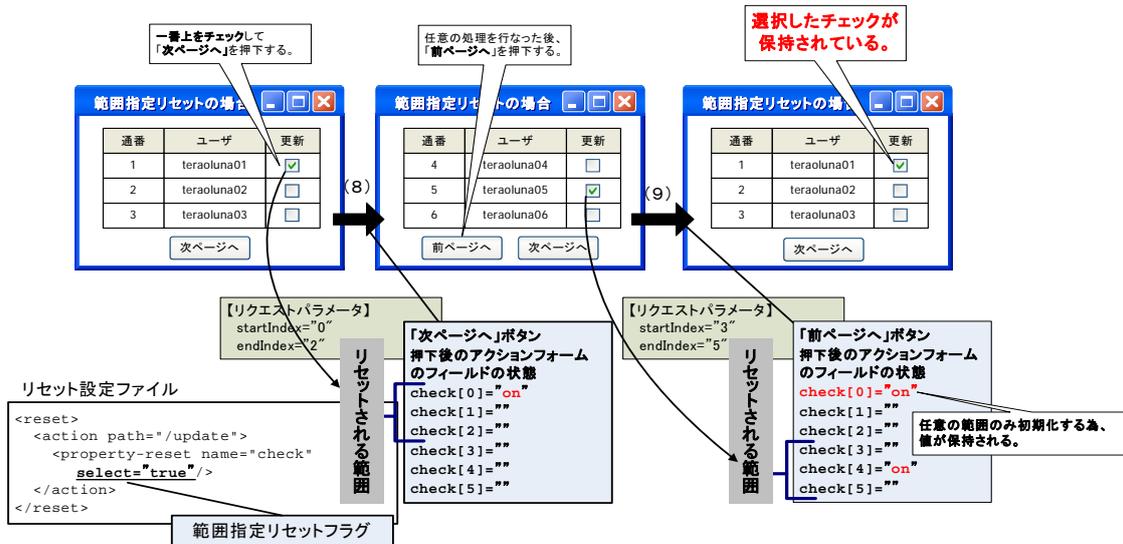
- (1) 【リセットを行わない場合】
チェックボックスの1つにチェックを付けてサブミットを行なう。リクエスト送信時には、チェックが付けられているフィールドのみが送信される。
(`check[0]="on"`が設定される。)
- (2) (1)にてチェックが付けられている箇所がアクションフォームに反映される。ここでは、チェック済みになっているフィールドのチェックを外し、代わりに未チェックだったチェックボックスにチェックを付けてサブミットを行なうと、`check[0]`、`check[1]`ともに"`on`"が設定されてしまう。
- (3) 【リセットを行なう場合】
アクションフォームのリセット機能を、リセット定義ファイル (`reset.xml`) ・ Struts 設定ファイル (`struts-config.xml`) で定義してサブミットを行なう。
- (4) Struts 設定ファイル (`struts-config.xml`) で、フォームサブミット時に用いられるアクションパス名と同じアクションパス名、及びリセット対象となるアクションフォームのフィールド名を、フォームリセット定義ファイル (`reset.xml`) に記述する。この設定によってアクションフォームの指定フィールドのみをリセットする。ここでは、`check` 配列は初期化される。
- (5) リセット後のアクションフォームフィールドに、リクエストパラメータで送信された値が格納される。これにより、直前の画面 (フォーム) の状態がアクションフォームに登録され、リクエストをまたがった不要な `check[0]="on"` の設定を消去することができる。

◆ 概念図

下記は ResetterImpl クラスを使用した場合の範囲指定リセットの例である。
複数ページの一覧画面で通常のリセット処理を行なうと...



複数ページの一覧画面で指定範囲リセット機能を使用すると...



◆ 解説

以下は操作対象のアクションフォームがセッションに格納されていることを前提とする。

- (6) 【リセットを行なう（指定範囲リセット機能—無効）場合】
アクションフォームのリセット機能を有効にして、サブミットを行なう。
- (7) 通常のリセット設定によってアクションフォームの指定フィールドをリセットすると、チェックボックスが複数画面にまたがる場合、全てのページのチェックボック

スが外されてしまう。

(8) 【リセットを行なう（指定範囲リセット機能—有効）場合】

アクションフォームのリセット機能を有効にし、かつ指定フィールドの指定範囲リセット機能を有効に設定して、サブミットを行なう。

- (9) 範囲指定リセットによってアクションフォームの指定フィールド（List または配列）の内の、任意の範囲のみをリセットすることが可能となる。これにより(7)のようにすべてのページのチェックボックスが外されるのではなく、各ページに表示されているチェックボックスの値を保持した状態でのリセット処理が可能となる。

■ 使用方法

◆ コーディングポイント

リセット設定ファイルは Struts のプラグイン機能を用いて初期化される。

- Struts 設定ファイル (struts-config.xml)

```
<struts-config>
.....
  <plug-in
className="jp.terasoluna.fw.web.struts.plugins.ResetterPlugIn">
  <set-property
    property="resetter"
    value="jp.terasoluna.fw.web.struts.reset.ResetterImpl"/>
  <set-property
    property="resources"
    value="/WEB-INF/reset.xml"/>
  <set-property
    property="digesterRules"
    value="/WEB-INF/reset-rules.xml"
  </plug-in>
.....
</struts-config>
```

TERASOLUNA Server Framework for Java (Web 版) 提供のプラグインクラスを設定する。

リセット処理を実行するクラス。

リセット処理の設定ファイル。アクションパスとリセット対象プロパティを紐付ける

リセット設定ファイルにはアクションパスとリセット対象のプロパティの紐付けを行う。

- リセット設定ファイル (reset.xml)

```
<reset>
```

```
  <action path="/resetAction">
```

```
    <property-reset name="field1" />
```

```
  </action>
```

```
  <action path="/selectResetAction">
```

```
    <property-reset name="field2" select="true" />
```

```
  </action>
```

```
</reset>
```

リセット処理を実行する対象のアクションパス。

初期化するアクションフォームのプロパティ名。
JXPathIndexedBeanWrapper の仕様に従い、ネストしたプロパティを記述可能。

範囲指定リセットを実行する場合は
select 属性を true に設定する。

範囲指定リセット処理を行う場合は、画面からリクエストパラメータとしてリセットする範囲のインデックス、startIndex と endIndex をサーバに送信する必要がある。

- 範囲指定リセットを実行する JSP の例

```
...
```

```
<ts:form action="/selectResetAction.do">
```

```
  ...
```

```
  <html:hidden property="startIndex" value="0" />
```

```
  <html:hidden property="endIndex" value="3" />
```

```
  ...
```

```
</ts:form>
```

```
...
```

配列、List の 0~3 番目の要素のみ初期化する。

<ts:pageLink>タグを使用する場合は、タグの機能で自動的に startIndex、endIndex を JSP に出力することが可能である。<ts:pageLink>の詳細は『WI-01 一覧表示機能』を参照のこと。

◆ 拡張ポイント

リセット処理の内容を変更したい場合は、Resetter インタフェースを実装したクラスを用意し、プラグインの設定ファイルに作成したクラス名を記述する。下記はパスが「~reset.do」の時にアクションフォームの value1、value2 フィールドを「true」に初期化する Resetter の例である。

- Resetter 実装クラスの例

```
public class MyResetter implements Resetter {  
  
    public void reset(FormEx form, ActionMapping mapping,  
        HttpServletRequest request) {  
        String path = request.getRequestURI();  
        if (path != null && path.endsWith("reset.do")) {  
            form.set("value1", "true");  
            form.set("value2", "true");  
        }  
    }  
}
```

Resetter インタフェースを実装し、reset メソッドにリセット処理を記述する。

- Struts 設定ファイル (struts-config.xml) の例

```
<plug-in  
    className="jp.terasoluna.fw.web.struts.plugins.ResetterPlugIn">  
    <set-property  
        property="resources"  
        value="/WEB-INF/reset.xml"/>  
    <set-property  
        property="resetter"  
        value="jp.terasoluna.sample.MyResetter"/>  
    .....
```

作成した Resetter 実装クラスを記述する

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.struts.plugins.ResetterPlugin	リセット設定ファイルを読み込み、リセット実行クラスを初期化するプラグイン。
2	jp.terasoluna.fw.web.struts.reset.Resetter	リセット処理を実行するインタフェース。
3	jp.terasoluna.fw.web.struts.reset.ResetterImpl	Resetter インタフェースのデフォルト実装クラス。設定ファイルをベースとし、リセット処理を自動的に行う。
4	jp.terasoluna.fw.web.struts.reset.ResetterResources	リセット設定ファイルを読み込んだ内容を保持するクラス。
5	jp.terasoluna.fw.web.struts.reset.ActionReset	リセット設定ファイルのアクションパス単位の設定内容を保持するクラス。
6	jp.terasoluna.fw.web.struts.reset.FieldReset	リセット設定ファイルのフィールド単位の設定内容を保持するクラス。

■ 関連機能

- 『WB-02 アクションフォーム拡張機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC12 フォームプロパティリセット」
 - ◇ /webapps/reset/*
 - ◇ /webapps/WEB-INF/reset/*
 - ◇ jp.terasoluna.thin.functionsample.reset.*

■ 備考

なし。

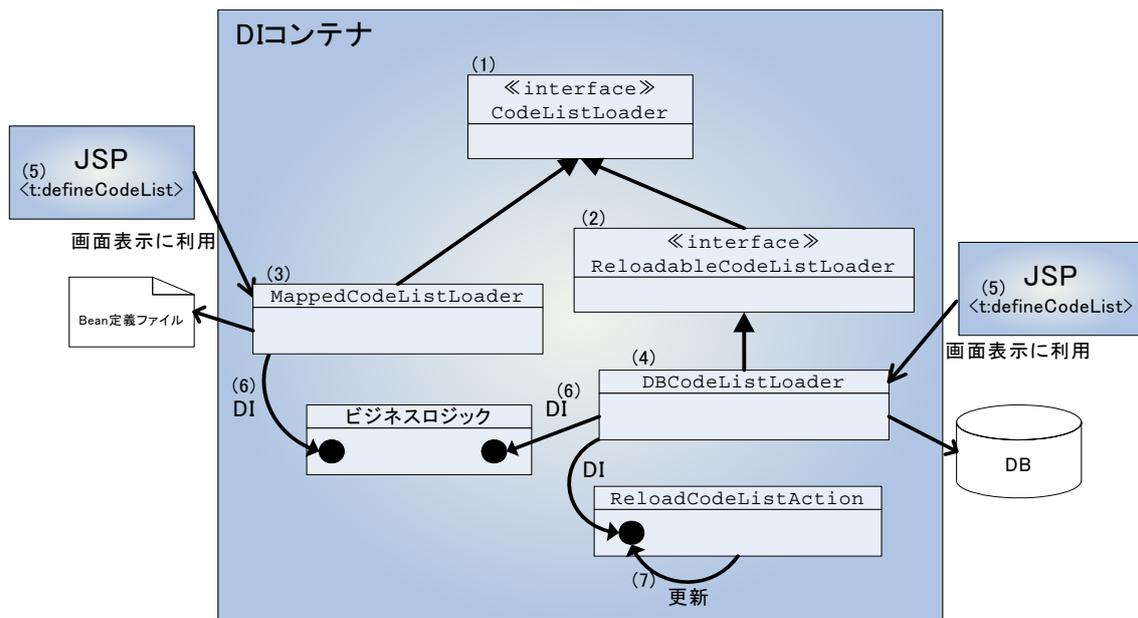
WB-05 コードリスト機能

■ 概要

◆ 機能概要

- アプリケーションから利用されるデータで、アプリケーション稼働中に更新されない（または、更新頻度が極めて低い）特定の意味を持つ名前と値のペアの集合をコードリストと呼ぶ。
- コードリストを Bean 定義ファイル、またはデータベースから読み込む機能である。
- データベースから読み込んだコードリストを更新できる。

◆ 概念図



◆ 解説

- (1) コードリストを読み込むクラスが実装するインタフェース `CodeListLoader` を提供する。
- (2) 更新可能なコードリスト読み込みクラスが実装するインタフェース `ReloadableCodeListLoader` を提供する。
- (3) Bean 定義ファイルからコードリストの設定を読み込むクラス、`MappedCodeListLoader` を提供する。
- (4) DB からコードリストの設定を読み込む更新可能なクラス、`DBCCodeListLoader` を提供する。
- (5) コードリストの情報を画面表示するために、`DefineCodeListTag` クラスを提供する。
※`DefineCodeListTag` クラスの詳細は『WJ-01～WK-06 画面表示機能』を参照のこと。
- (6) ビジネスロジックからコードリストの情報を参照する場合は、DI コンテナの機能を用い、ビジネスロジックインスタンスに `CodeListLoader` 実装クラスのインスタンスを設定する。
- (7) コードリストの更新を行う場合は、`ReloadCodeListAction` クラスに更新対象の `ReloadableCodeListLoader` インスタンスを設定し、`doExecute()`メソッドを実行する。

■ 使用方法

◆ コーディングポイント

【Bean 定義ファイルを用いたコードリストの初期化】

Bean 定義ファイルからコードリストの情報を読み込む場合は、Bean 定義ファイル内にコードリストの情報を直接記述する。但し、記述をおこなう Bean 定義ファイルは ContextLoaderListener によって読み込まれる Bean 定義ファイルでなければならない。(プラグインからロードされる Bean 定義ファイルに定義してはならない)

- デプロイメントディスクリプタ (web.xml)

```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>
```

ContextLoaderListener から読み込まれるコードリストを定義する対象の Bean 定義ファイル

- Bean 定義ファイル

```
<bean id="sampleCodeList"
class="jp.terasoluna.fw.web.codelist.MappedCodeListLoader"
init-method="load">
<property name="codeListMap">
<map>
<entry key="001">
<value>value001</value>
</entry>
<entry key="002">
<value>value002</value>
</entry>
<entry key="003">
<value>value003</value>
</entry>
</map>
</property>
</bean>
```

class 属性に MappedCodeListLoader を指定し、必ず init-method で load を指定する。

コードリスト情報は Map 型で codeListMap 変数に設定する。コードリストのキー名は entry 要素、値は value 要素に設定する。

【データベースを用いたコードリストの初期化】

データベースからコードリストの情報を読み込む場合は、コードリスト情報を取得する SQL 文を Bean 定義ファイルから設定する。

コードリストを取得する SQL 文は自由に設定できるが、データベースから取得した結果の 1 カラム目がコードリストのキーの値、2 カラム目がコードリストの値に自動的に設定される。Bean 定義ファイルから読み込むコードリスト同様、ContextLoaderListener によって読み込まれる Bean 定義ファイルに設定すること。

● Bean 定義ファイル

```
<bean id="sampleDBCodeList"
      class="jp.terasoluna.fw.web.codelist.DBCodeListLoader"
      init-method="load">
  <property name="dataSource">
    <ref bean="TerasolunaDataSource"/>
  </property>
  <property name="sql">
    <value>SELECT KEY, VALUE FROM CODE_LIST</value>
  </property>
</bean>
```

class 属性に DBCodeListLoader を指定し、必ず init-method で load を指定する。

dataSource プロパティにはコードリスト情報を読み込む対象のデータソースを指定。sql にはコードリストを取得する SQL 文を設定する。この例では KEY がコードリストの id、VALUE がコードリストの値となる。

【コードリストを画面表示に利用する】

コードリストは画面表示に利用されることを本来の目的として提供されている。
以下に JSP でのコードリストの利用方法の例を示す。

● JSP

```
<%@ taglib prefix="t" uri="/WEB-INF/terasoluna.tld"%>
```

```
<html><head><body>
```

```
<html:form action="/codeList">
```

```
<t:defineCodeList id="sampleCodeList"/>
```

```
<html:select property="codeListId">
```

```
  <html:options collection="sampleCodeList" property="id"
```

```
    labelProperty="name" />
```

```
</html:select>
```

```
データ : <t:writeCodeCount id="sampleCodeList" />
```

```
</html:form></body></html>
```

コードリストを page スコープに定義するタグ、
<defineCodeList>タグ。id には CodeListLoader の
BeanId を指定し、その値がそのままページ内変
数名となる。

コードリストの件数を出力する、
<writeCodeCount>タグ。画面にコード
リストの件数を直接出力する。



【ビジネスロジックでコードリストを参照する】

ビジネスロジックでコードリストを参照する場合は、ビジネスロジック実装クラスに `CodeListLoader` インスタンスへの参照を持たせる。TERASOLUNA Server Framework for Java (Web 版)が提供する `MappedCodeListLoader`、`DBCodeListLoader` は Spring フレームワークの DI の機能を使用することを推奨する。

● Bean 定義ファイル

```
<bean id="sampleCodeList"
class="jp.terasoluna.fw.web.codelist.MappedCodeListLoader"
.....
/>
<bean id="sampleBusinessLogic"
class="jp.terasoluna.sample.SampleBusinessLogic">
  <property name="codeListLoader">
    <ref bean="sampleCodeList"/>
  </property>
</bean>
```

ビジネスロジック実装クラスに `CodeListLoader` インスタンスを設定する。

● ビジネスロジック実装クラス

```
public class SampleBusinessLogic {
  private CodeListLoader codeListLoader = null;
  public void setCodeListLoader(CodeListLoader codeListLoader) {
    this.codeListLoader = codeListLoader;
  }

  public void someBusiness() {
    CodeBean[] beans = codeListLoader.getCodeBeans();
    .....
  }
}
```

`CodeListLoader` 型の属性とアクセサメソッドを定義。

コードリストの情報は `CodeBean` クラスで実装されている。`getCodeBeans()`メソッドで、`CodeBean` の配列が取得できる。

※ビジネスロジックではコードリスト情報の参照だけを許可する。コードリストは原則としてアプリケーション稼働中は不変の情報であるため、ビジネスロジックからコードリストの内容が変更されることのないように注意すること。

【コードリストの更新】

更新可能なコードリストインタフェース **ReloadableCodeListLoader** を実装した **DBCodeListLoader** はアクションクラス経由で再度データベースから値を取得し、コードリスト情報の更新をすることが可能である。更新処理を行う場合は、**ReloadCodeListAction** を定義するだけでよい。

コードリストの更新時はアプリケーションを閉塞状態にするなど、ユーザがアクセスできない状態で更新しなければならない。また、コードリストの状態保持は DI コンテナ毎に管理されるため、クラスタ環境等、複数の JVM が動作する環境では各 JVM において更新処理が必要である。

● Bean 定義ファイル

```
<bean id="sampleDBCodeList"
  class="jp.terasoluna.fw.web.codelist.DBCodeListLoader"
  init-method="load">
  .....
</bean>
<bean name="/reloadAction" scope="prototype"
  class="jp.terasoluna.fw.web.struts.actions.ReloadCodeListAction">
  <property name="codeListLoader"
    <ref bean="sampleDBCodeList"/>
  </property>
</bean>
```

アクションのクラスには **ReloadCodeListAction** を指定。

◆ 拡張ポイント

`CodeListLoader` インタフェースを実装することで、独自のコードリスト読み込みクラスを定義することが可能である。

- `CodeListLoader.java`

```
public interface CodeListLoader {  
  
    void load();  
  
    CodeBean[] getCodeBeans();  
  
}
```

`load()`メソッドではコードリスト情報の初期化を行う。アプリケーションは初期化時にこの `load()`メソッドを呼び出し、コードリスト情報の初期化を行わなければならない。

`getCodeBeans()`メソッドはコードリストを実装した `CodeBean` クラスの配列を返却する。コードリストは原則としてアプリケーションで不変の情報であるため、このメソッドを使用するクラスで参照が変更されないように注意する必要がある。

更新可能なコードリストは `ReloadableCodeListLoader` インタフェースを実装する。

- `ReloadableCodeListLoader.java`

```
public interface ReloadableCodeListLoader extends CodeListLoader {  
  
    void reload();  
  
}
```

`reload()`メソッドはコードリスト情報の更新を実行する。`ReloadableCodeListLoader` インタフェースを実装していれば、`ReloadCodeListAction` 経由でコードリストの更新を実行することも可能である。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.codelist.CodeListLoader	コードリストを読み込むクラスが実装するインタフェース
2	jp.terasoluna.fw.web.codelist.ReloadableCodeListLoader	更新可能なコードリストを読み込むクラスが実装するインタフェース
3	jp.terasoluna.fw.web.codelist.MappedCodeListLoader	Bean 定義の設定を元にコードリストを読み込む CodeListLoader 実装クラス。
4	jp.terasoluna.fw.web.codelist.DBCodeListLoader	データベースからコードリストを読み込む ReloadableCodeListLoader 実装クラス。
5	jp.terasoluna.fw.web.codelist.DBCodeListQuery	DBCodeListLoader クラスがデータベースにアクセスするために使用するクラス。
6	jp.terasoluna.fw.web.codelist.CodeBean	1件のコードリスト情報を実装するクラス。
7	jp.terasoluna.fw.web.taglib.DefineCodeListTag	コードリストをページ変数に定義するタグ。
8	jp.terasoluna.fw.web.taglib.WriteCodeCountTag	コードリストの件数をページに出力するタグ。
9	jp.terasoluna.fw.web.struts.actions.ReloadCodeListAction	ReloadableCodeListLoader の更新処理を実行するアクションクラス。

■ 関連機能

- 『WE-04 コードリスト再読み込み機能』
- 『WJ-10 コードリスト定義機能』
- 『WJ-11 コードリスト件数出力』
- 『WJ-12 指定コードリスト値表示機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC13 コードリスト」
 - ◇ /webapps/codelist/*
 - ◇ /webapps/WEB-INF/codelist/*
 - ◇ jp.terasoluna.thin.functionsample.codelist.*

■ 備考

なし。

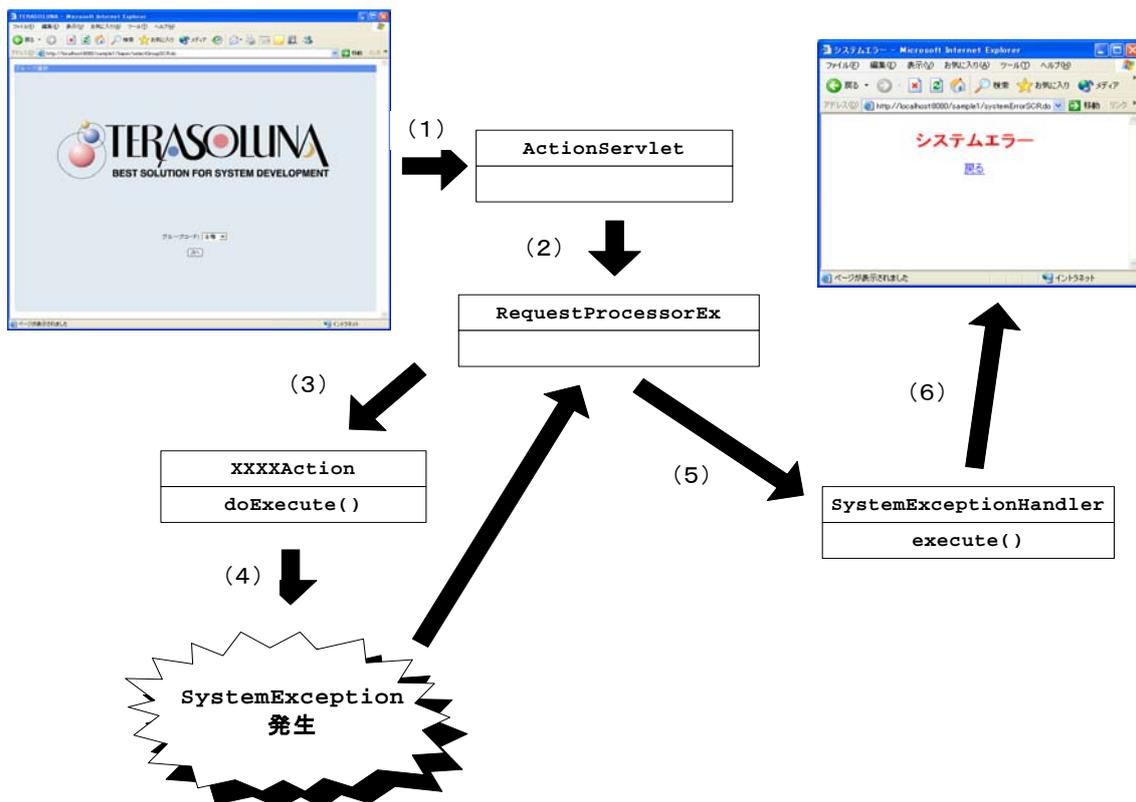
WC-01 Action 実行時システム例外処理機能

■ 概要

◆ 機能概要

- Action 実行時（ビジネスロジックを含む）にシステム例外が発生した場合に、共通的なシステム例外処理を提供する。
- システム例外処理は、以下の処理を行なう。
 - スタックトレースをログに出力する。
 - システム例外発生時のエラー画面に遷移させる。

◆ 概念図



◆ 解説

以下の解説は Struts 設定ファイル(struts-config.xml)に SystemExceptionHandler の設定が正しくされている事が前提となる。設定方法の詳細は、コーディングポイントを参照のこと。

- (1) ユーザからリクエストが送信される。
- (2) ActionServlet により、RequestProcessorEx が呼び出される。
- (3) RequestProcessorEx により、リクエストのアクションパスに該当する、ActionEx#doExecute()メソッドが実行される。
- (4) アクション内部でシステム例外が発生する。
- (5) 例外がキャッチされ、システム例外を処理するクラス (SystemExceptionHandler#execute()メソッド) が呼び出される。
- (6) SystemExceptionHandler#execute()メソッドでは、発生したシステム例外に格納されたメッセージキーからエラーメッセージを取得し、スタックトレースをログに出力する。また、システム例外も、遷移先の JSP で利用することを想定し、PageContext.EXCEPTION をキーにしてリクエストに格納し、Struts 設定ファイル (struts-config.xml) で指定された<exception>要素の path 属性で指定されたアクションパスに遷移する (ここではシステムエラー画面に遷移している)。

■ 使用方法

◆ コーディングポイント

システム例外発生時のログ出力、及び遷移先パス指定は、Struts 設定ファイル (struts-config.xml) の<exception>要素に指定する。

● Struts 設定ファイル(struts-config.xml)

```

<struts-config>
  ...
  <global-exceptions>
    <exception key="errors.E001" path="/systemError.do"
      className="jp.terasoluna.fw.web.struts.action.ExceptionConfigEx"
      handler="jp.terasoluna.fw.web.struts.action.SystemExceptionHandler"
      type="jp.terasoluna.fw.exception.SystemException" >
      <set-property property="module" value="/sub" />
    />
  </global-exceptions>
  ...
  <action path="/cancelServerBlockageAction"
    name="_cancelServerBlockageForm"
    type="jp.terasoluna.sample1.common.web.action.ServerBlockageAction">
    <exception key="errors.E002" path="/cancelServerBlockageForm.do"
      className="jp.terasoluna.fw.web.struts.action.ExceptionConfigEx"
      handler="jp.terasoluna.fw.web.struts.action.SystemExceptionHandler"
      type="jp.terasoluna.fw.exception.SystemException" />
    <forward name="failure" path="/cancelServerBlockageSCR.do" />
    <forward name="success" contextRelative="true" redirect="true"
      path="/logon/logonSCR.do" />
  </action>
  ...
</struts-config>

```

type 属性に SystemException を指定する。

<htmlx:error>、<html:error>タグによって、JSP にエラーメッセージを表示するメッセージリソースのキーを指定する。

handler には必ず SystemExceptionHandler を指定する。未指定時にはエラーにならないが、Struts の ExceptionHandler が実行される。

className 属性には、ExceptionConfigEx を指定する。この指定が無い場合は、遷移先がコンテキスト相対となる。なお ExceptionConfigEx を指定した場合、遷移先モジュールを指定できる。

遷移先のパスを指定する。

<action>要素内に<exception>要素を定義した場合、<global-exceptions>要素の定義より優先される。

遷移先のモジュールが指定されない場合、遷移先はモジュール相対（現在のモジュール内の遷移）となる。

◆ 拡張ポイント

- なし

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.struts.action.SystemExceptionHandler	システム例外を処理するハンドラクラス。
2	jp.terasoluna.fw.exception.SystemException	TERASOLUNA Server Framework for Java から発生する、システム例外クラス。

■ 関連機能

- なし

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC14 アクション実行時システム例外処理」
 - ◇ /webapps/exception/*
 - ◇ /webapps/WEB-INF/exception/*
 - ◇ jp.terasoluna.thin.functionsample.exception.*
- TERASOLUNA Server Framework for Java (Web 版) チュートリアル
 - 「2.9 例外処理」
 - /webapps/WEB-INF/struts-config.xml

■ 備考

- なし

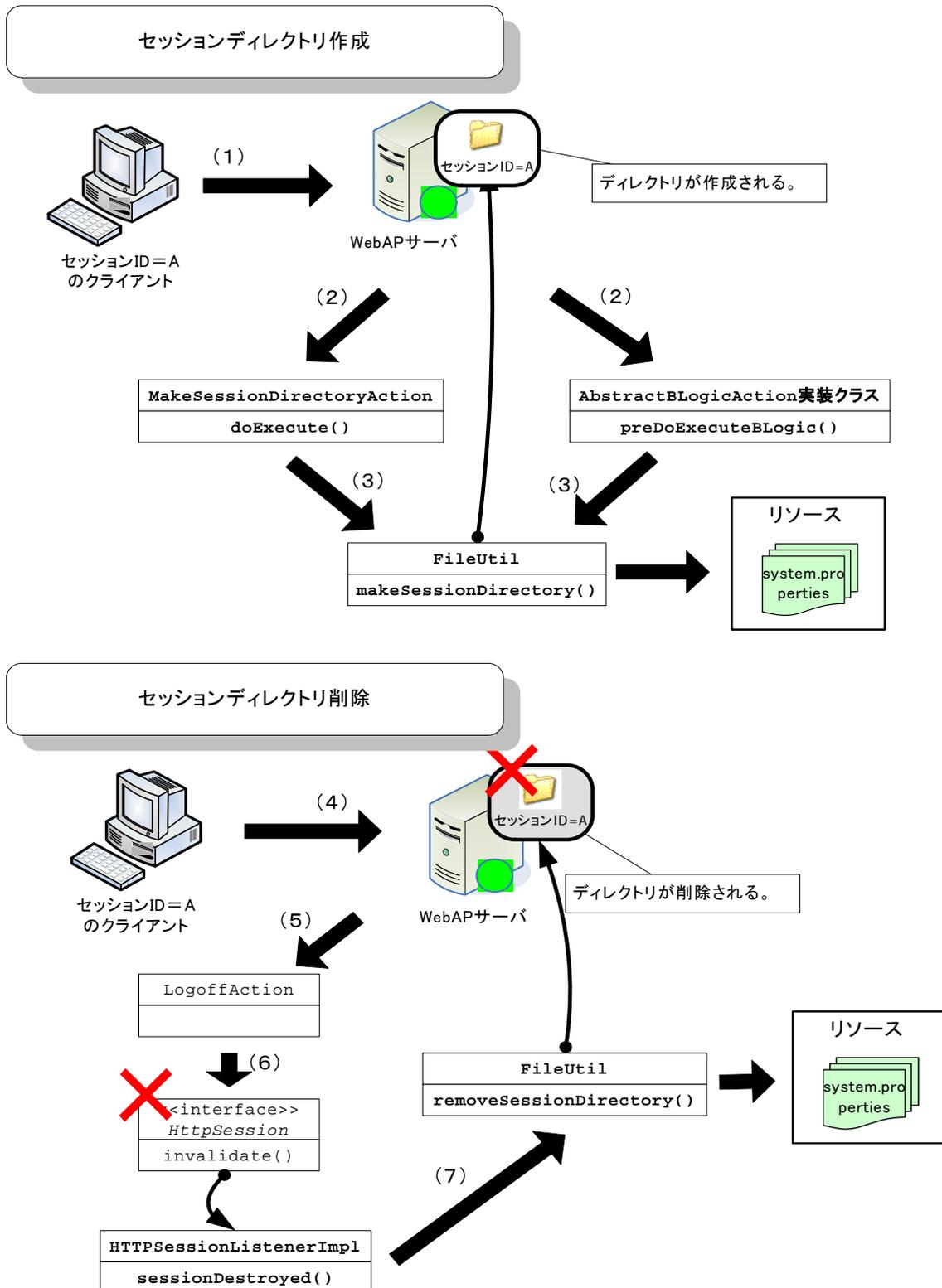
WD-01 セッションディレクトリ機能

■ 概要

◆ 機能概要

- サーバサイドで生成された PDF ファイルなどを格納するための一時ディレクトリ（以降、セッションディレクトリ）をログオンユーザ毎に作成する。また、ログオフ時には一時ディレクトリを削除する。
- セッションディレクトリの作成と、ログオフ時のセッションディレクトリ破棄を行う。
- セッションディレクトリの作成には以下の二つの方法で利用可能である。
 - MakeSessionDirectoryAction の使用
 - ☆ 『WE-06 セッションディレクトリ作成機能』を参照のこと。
 - アクションクラスで直接セッションディレクトリ作成
- セッションディレクトリの破棄は以下の方法で利用可能である。
 - LogoffAction クラスの使用
 - ☆ 『WE-08 ログオフ機能』を参照のこと。

◆ 概念図



◆ 解説

- (1) ログオン時など、ユーザからリクエストが送信される。
- (2) セッションディレクトリ作成用のアクションが実行され、`FileUtil#makeSessionDirectory()`メソッドを呼び出す。アクションクラスには Terasoluna Server Framework for Java (Web 版) が提供する `MakeSessionDirectoryAction`、または業務開発者により実装された `AbstractBLogicAction` 実装クラスを指定する。
- (3) `FileUtil` にて、セッション ID に応じたセッションディレクトリが作成される。
- (4) ログオフ時に、ユーザからリクエストが送信される。
- (5) ログオフ処理のアクションクラスである `LogoffAction` が実行される。
- (6) `LogoffAction` により、セッションが無効化されると、`HttpSessionListenerImpl#sessionDestroyed()`メソッドのコールバックが行われる。
- (7) `sessionDestroyed()`メソッドにより、セッションディレクトリが削除される。

■ 使用方法

◆ コーディングポイント

1. プロパティ設定

セッションディレクトリが配置されるベースとなるディレクトリを決定し、システム設定プロパティファイル (`system.properties`) に設定する。

- システム設定プロパティファイル (`system.properties`)

```
session.dir.base=/tmp/sessions
```

/tmp/sessions 配下に、セッションディレクトリが格納される。

2. セッションディレクトリ作成

セッションディレクトリ作成には以下の二つの方法で利用可能である。

(ア) `MakeSessionDirectoryAction` の使用

TERASOLUNA Server Framework for Java (Web 版)が提供するセッションディレクトリ作成クラスのアクションマッピング定義を設定する。

- Struts 設定ファイル (`struts-config.xml`)

```
<action path="/makeSessionDir"  
  scope="session"  
  parameter="/foo.jsp">;  
</action>
```

同名で指定する。
任意の名前でよい。

- Bean 定義ファイル

```
<bean name="/makeSessionDir" scope="prototype"  
  class="jp.terasoluna.fw.web.struts.actions.MakeSessionDirectoryAction">  
</bean>
```

TERASOLUNA が提供するセッションディレクトリ作成クラス。

(イ) アクションクラスで直接セッションディレクトリ作成

ビジネスロジック実行前に呼び出される `AbstractBLogicAction#preDoExecuteBLogic()` にセッションディレクトリ作成を実装する。

➤ `AbstractBLogicAction` の実装クラス

```
package jp.terasoluna.sample1.actions;
public class SampleAction extends AbstractBLogicAction {

    public void preDoExecuteBLogic(HttpServletRequest req,
        HttpServletResponse res,
        P params)
        throws Exception {
        .....
        HttpSession session = (HttpSession) request.getSession();
        boolean result = FileUtil.makeSessionDirectory(session.getId());
        .....
    }
}
```

AbstractBLogicActionを継承する。

ビジネスロジックが実行される前に呼び出されるメソッド。

セッションを取得後、セッション ID を引数に `FileUtil#makeSessionDirectory(String sessionId)` を実行する。

Struts 設定ファイルと Bean 定義ファイルにアクションマッピング定義を設定する。

➤ Struts 設定ファイル (struts-config.xml)

```
<action path="/makeSessionDir"
    name="_sampleForm"
    scope="session">
    <forward name="success" path="/logonSCR.do"/>
    <forward name="failure" path="/errRedirect.do"/>
</action>
```

同名で指定する。
任意の名前でよい。

➤ Bean 定義ファイル

```
<bean name="/makeSessionDir" scope="prototype"
    class="jp.terasoluna.sample1.actions.SampleAction">
    <property name="sampleBLogic">
        <ref bean="SampleBLogic"></ref>
    </property>
</bean>
<bean id="SampleBLogic"
    class="jp.terasoluna.sample1.blogic.SampleBLogic">
</bean>
```

上記で実装した、セッションディレクトリ生成用アクションクラスを指定する。

3. セッションディレクトリの削除

デプロイメントディスクリプタに、TERASOLUNA Server Framework for Java (Web版)が提供する HTTP セッションライフサイクルイベントの処理クラス `HttpSessionListenerImpl` の使用を宣言する。

➤ デプロイメントディスクリプタ (web.xml)

```
<web-app>
  .....
  <listener>
    <listener-class>
      jp.terasoluna.fw.web.HttpSessionListenerImpl
    </listener-class>
  </listener>
  .....
</web-app>
```

セッションの生成、無効化時に起動するリスナークラスを指定する。

Struts 設定ファイルと Bean 定義ファイルにログオフ時のアクションマッピング定義を設定する。

LogoffAction の詳細は『WE-08 ログオフ機能』を参照のこと。

➤ Struts 設定ファイル (struts-config.xml)

```
<action path="/logoff"
  scope="session"
  parameter="/foo.jsp">;
</action>
```

ログオフ時にセッションが無効化され、セッションディレクトリが削除される。

➤ Bean 定義ファイル

```
<bean name="/logoff" scope="prototype"
  class="jp.terasoluna.fw.web.struts.actions.LogoffAction">
</bean>
```

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.util.FileUtil	ファイル・ディレクトリの生成、削除、取得を行うユーティリティクラスである。詳細は『CD-01 ユーティリティ機能』を参照のこと。
2	jp.terasoluna.fw.web.struts.actions.MakeSessionDirectoryAction	セッションディレクトリをログオンユーザ毎に作成する。詳細は『WE-06 セッションディレクトリ作成機能』を参照のこと。
3	jp.terasoluna.fw.web.struts.actions.LogoffAction	ログオフ時に実行されるアクションクラスである。ログオフ処理として、セッションの無効化を指定する。詳細は『WE-08 ログオフ機能』を参照のこと。
4	jp.terasoluna.fw.web.HttpSessionListenerImpl	セッションの生成、無効化時の動作を定義する HttpSessionListener インタフェースを実装したクラスである。
5	javax.servlet.http.HttpSessionListener	セッションの生成、無効化時の動作を定義するインタフェースである。

◆ 拡張ポイント

- 上記『コーディングポイント』『アクションクラスで直接セッションディレクトリ作成』の `AbstractBLogicAction` を拡張することにより、セッションディレクトリ作成の拡張を行うことが可能である。
- 上記『コーディングポイント』『セッションディレクトリの削除』で使用されている、TERASOLUNA Server Framework for Java (Web 版)提供の HTTP セッションライフサイクルイベントの処理クラス `HttpSessionListenerImpl` を使用せず、インタフェースを実装したクラスをデプロイメントディスクリプタに設定することにより、セッションディレクトリ削除の拡張を行うことが可能である。

■ 関連機能

- 『CD-01 ユーティリティ機能』
- 『WE-06 セッションディレクトリ作成機能』
- 『WE-08 ログオフ機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC15 セッションディレクトリ」
 - ◇ /webapps/sessiondir/*
 - ◇ /webapps/WEB-INF/sessiondir/*
 - ◇ jp.terasoluna.thin.functionsample.sessiondir.*

■ 備考

- なし

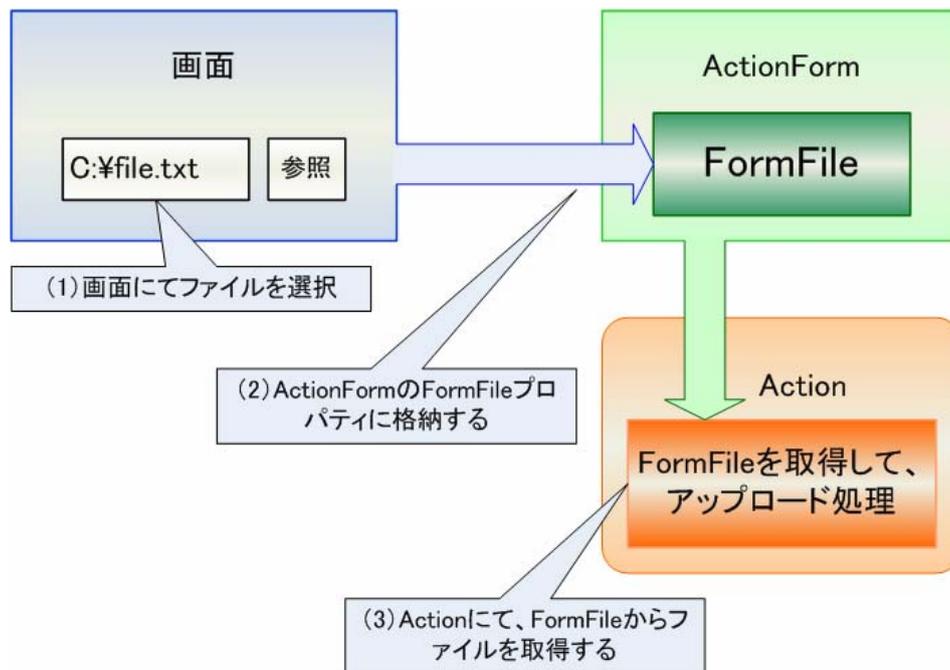
WD-02 ファイルアップロード機能

■ 概要

◆ 機能概要

- Webブラウザからファイルをアップロードする機能を提供する。
 - Jakarta Commons FileUpload および Struts FormFile の機能をほぼそのまま使用する。

◆ 概念図



◆ 解説

- (1) <html:file>要素でファイルの入力フィールドを作成し、アップロードするファイルを選択する。
- (2) 選択されたファイルがアップロードされて、ActionFormのFormFileのプロパティに格納される。
- (3) 格納されたFormFileをActionで取得して、ファイルへの出力処理やデータベースへの登録処理を行う。

■ 使用方法

◆ コーディングポイント

- Struts 設定ファイル
 - <controller>要素の設定
 - ◇ bufferSize 属性の指定
bufferSize 属性にはアップロードファイルの内容を読み込むときに一度にメモリにどれだけのサイズを読み込むかを指定する。デフォルトは 4096 バイトである。
 - ◇ maxFileSize 属性の指定
アップロードできるファイルサイズの上限を指定する。「K」、「M」、「G」を単位として使用可能である。例) 100M→100 メガバイトがアップロードできるファイルの最大のサイズ
 - ◇ memFileSize 属性の指定
Struts ではアップロードされたファイルの内容をバイト配列としてメモリに置くか、一時ファイルとしてハードディスクに保存するかを選択できる。memFileSize 属性にはメモリ保存とハードディスク保存の閾値を指定する。maxFileSize 同様、「K」、「M」、「G」が単位として使用可能である。
 - ◇ tempDir 属性の指定
アップロードされたファイルを一時ファイルとして保存する場合、保存されるディレクトリ名を指定する。
 - JSP
 - <ts:form>要素の設定
 - ◇ enctype 属性の指定
enctype 属性には、以下のように multipart/form-data を指定する。

```
<ts:form action="/fileup" enctype="multipart/form-data">
```
 - <html:file>要素の設定
 - ◇ property 属性の指定
property 属性には、アクションフォームの FormFile のプロパティを指定する。

```
<html:file name="dynaFormBean" property="fileup" />
```
 - アクションフォーム
 - Struts 設定ファイル (DynaValidatorActionFormEx を使用)
 - ◇ <form-property>要素の type 属性に FormFile を指定する。

```
<form-property name="fileup"  
type="org.apache.struts.upload.FormFile"/>
```

- アクションフォームクラス (ValidatorActionFormEx を使用)
 - ◇ FormFile のプロパティおよび Getter/Setter を作成する。

```
private FormFile fileup = null;
public FormFile getFileup() {
    return fileup;
}
public void setFileup(FormFile fileup) {
    this.fileup = fileup;
}
```

- アクション

- アクションでは、Struts によって設定された FormFile を使用して、ファイルを出力する。以下の記述は例であり、ファイルの出力方式および出力場所については別途検討する必要がある。

```
//アップロードファイルの取得
FormFile fileup = (FormFile) dynaForm.get("fileup");

//getInputStreamメソッドを使用し、入力ストリームを取得
InputStream is = fileup.getInputStream();

//入力ストリームをバッファリング
BufferedInputStream inBuffer = new BufferedInputStream(is);

//ファイルのアップロード先を指定して、出力ストリームを生成
FileOutputStream fos = new FileOutputStream
    ("c:/tmp/" + fileup.getFileName());
//出力ストリームをバッファリング
BufferedOutputStream outBuffer = new BufferedOutputStream(fos);

//入力データがなくなるまで入出力処理を実行
int data = 0;
while ((data = inBuffer.read()) != -1) {
    outBuffer.write(data);
}

//解放
outBuffer.flush();
inBuffer.close();
outBuffer.close();
//一時領域のアップロードデータを削除
fileup.destroy();
```

- 出力場所の指定
上記の例では、“c:/tmp/” を指定して出力しているが、プロパティファイルなどを使用して指定する方法を推奨する。また、一時的なファイルの出力であれば TERASOLUNA Server Framework for Java (Web 版)の『WD-01 セッションディレクトリ機能』を使用することで、ディレクトリの作成および削除を簡単に行うことができる。

◆ 拡張ポイント

なし。

■ 関連機能

- 『WD-03 ファイルダウンロード機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「WD-02 ファイルアップロード機能」
 - ◇ /webapps/upload/*
 - ◇ /webapps/WEB-INF/upload/*
 - ◇ jp.terasoluna.thin.functionsample.upload.*

■ 備考

なし。

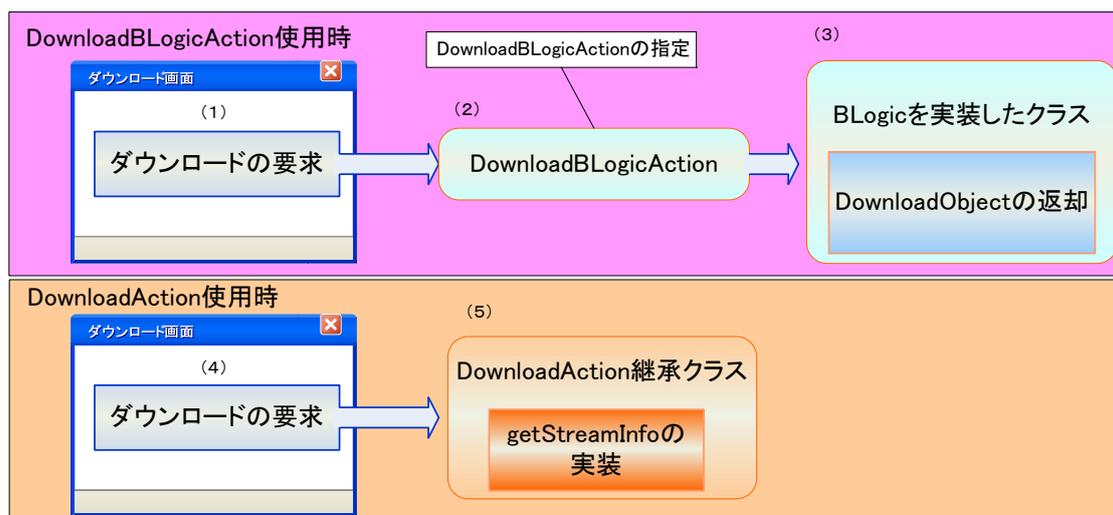
WD-03 ファイルダウンロード機能

■ 概要

◆ 機能概要

- クライアントへファイルをダウンロードする機能を提供する。
ダウンロードの実装は Terasoluna Server Framework for Java (Web 版) に依存した DownloadBLogicAction を利用する方法と Struts から提供されている DownloadAction を継承する方法の2つがある。
2つの実装でやれることに明確な差は無く、業務ロジックを BLogic に統一したい場合に DownloadBLogicAction を利用する。

◆ 概念図



◆ 解説

- DownloadBLogicAction を利用する場合
 - (1) ダウンロードするファイルの要求を画面や別クラスから要求をする。
 - (2) DownloadBLogicAction を Bean 定義ファイルで指定する。
 - (3) BLogic を実装したクラスで AbstractDownloadObject のサブクラスを返却する。
 - AbstractDownloadObject サブクラス一覧

	クラス名	概要
1	jp.terasoluna.fw.web.struts.actions.DownloadFile	ファイルをダウンロード対象とするためのクラス
2	jp.terasoluna.fw.web.struts.	インプットストリームをダウンロード対象とするためのクラス

	actions.DownloadInputStream	
3	jp.terasoluna.fw.web.struts.actions.DownloadString	String をダウンロード対象とするためのクラス
4	jp.terasoluna.fw.web.struts.actions.DownloadByteArray	バイト配列をダウンロード対象とするためのクラス

- DownloadAction を利用する場合
- (4) ダウンロードするファイルの要求を画面や別クラスから要求をする。
 - (5) DownloadAction を継承したクラスの getStreamInfo メソッドを呼び出して、ファイルのダウンロード処理を行う。

■ 使用方法

◆ コーディングポイント

- DownloadBLogicAction を指定してファイルダウンロードを行う場合
ファイル名のデフォルトエンコードの実装は IE のみに対応しており、Firefox 等のブラウザでは文字化けが発生する。クロスブラウザの対応は拡張ポイント「クロスブラウザに対するファイル名の文字化け回避方法」を参照のこと。

➤ Bean 定義ファイルで DownloadBLogicAction の指定を行う

```
<bean name="/downloadBLogic"
      class="jp.terasoluna.fw.web.struts.actions.DownloadBLogicAction"
      scope="singleton">
  <property name="businessLogic" ref="downloadBLogic"/>
</bean>
<bean id="downloadBLogic"
      class="jp.sample.project.blogic.DownloadBLogic" scope="singleton"/>
```

➤ BLogic インターフェースを実装する

BLogic インターフェース実装クラスでは、ダウンロードファイルの種別に応じた AbstractDownloadObject のサブクラスを BLogicResult に設定する。

以下に DownloadFile を使用した例を記述する。

例では BLogicResult に直接 DownloadFile を格納しているが、DTO 内にネストした形でもダウンロード処理は行われる。

```
public class DownloadBLogic implements BLogic {

    public BLogicResult execute(DownloadInput param) {
        BLogicResult result = new BLogicResult();

        File file = new File("filepath¥¥download.txt");
        DownloadFile downloadFile = new DownloadFile(file);
        result.setResultObject(downloadFile);

        return result;
    }
}
```

- DownloadAction を継承してファイルダウンロードを行う場合

DownloadAction#getStreamInfo()の実装を行う

➤ StreamInfo オブジェクトの返却

- ✧ getStreamInfo()は StreamInfo オブジェクトを返却する必要がある。
StreamInfo は DownloadAction の内部インタフェースであり、

DownloadAction には StreamInfo インタフェースを実装した FileStreamInfo クラスを提供している。

➤ FileStreamInfo の使用

FileStreamInfo を使用する場合は以下のように、ダウンロードするファイルのコンテンツタイプと、File オブジェクトが必要となる。以下に FileStreamInfo を使用した例を記述する。

```
/**
 * getStreamInfo()の実装例。
 */
protected StreamInfo getStreamInfo(ActionMapping mapping,
    ActionForm form, HttpServletRequest request,
    HttpServletResponse response) throws Exception {

    //ファイル名の取得
    String fileName = new String(
        bean.getFileName().getBytes(encoding), "ISO-8859-1");

    //コンテンツタイプの取得
    String contentType = bean.getFileContentType();

    //レスポンスヘッダーの設定
    response.setHeader("Content-disposition",
        "attachment; filename=" + fileName);

    //ダウンロードファイルの生成
    File file = new File("c:/tmp/" + fileName);

    //FileStreamInfoの返却
    return new FileStreamInfo(contentType, file);
}
```

➤ 固定環境のダウンロードファイル名対応

エンドユーザが利用するブラウザや OS が固定である場合、以下のようにすることで文字化けを回避することができる。

getBytes()の引数には、“Windows-31J”など固定の環境にあった適切な文字コードを指定する。クロスブラウザの対応は拡張ポイント「クロスブラウザに対するファイル名の文字化け回避方法」を参照のこと。

```
String fileName = new String(
    bean.getFileName().getBytes(encoding), "ISO-8859-1");
```

➤ Content-disposition ヘッダの指定

- ◇ ダウンロード時のファイル名は Content-disposition ヘッダの設定を行う必要がある。今回の例ではヘッダの設定は、getStreamInfo()メソッドの中で設定をする。

ファイルをブラウザで開かずに、ダウンロードをさせるか否かを判断させる場合は、“attachment”を指定する。“inline”を指定するとファイルをブラウザで開くことになる。

```
response.setHeader("Content-disposition",  
    "attachment; filename=" + fileName);
```

◆ 拡張ポイント

- クロスブラウザに対するファイル名の文字化け回避方法
IE と Firefox などブラウザの差異により、Content-disposition ヘッダに設定したファイル名が文字化けを起こす。原因は、IE はファイル名を UTF-8 で表記したものを x-www-form-url 符号化する仕様になっているが、Firefox は RFC2047 の規定に従っているためである。
クロスブラウザに対応するためには、Http ヘッダの User-Agent によりファイル名をブラウザ毎にエンコードする必要がある。
以下に IE と Firefox に対応する例を記載する。
 - DownloadBLogicAction を指定してファイルダウンロードを行う場合
DownloadFileNameEncoder を実装し、アプリケーション起動時に jp.terasoluna.fw.web.struts.actions.FileDownloadUtil#setEncoder で設定を行う。
実装例は Firefox と判断した場合、Jakarta プロダクトの commons-codec を利用しエンコードを行っている。

```
public class MyEncoder implements DownloadFileNameEncoder {  
  
    public String encode(String original, HttpServletRequest request,  
        HttpServletResponse response) {  
        String userAgent = request.getHeader("User-Agent");  
        // IE の場合  
        if (StringUtils.contains(userAgent, "MSIE")) {  
            return encodeForIE(original);  
        }  
        // Firefox の場合  
        } else if (StringUtils.contains(userAgent, "Gecko")) {  
            return encodeForGecko(original);  
        }  
        return encodeForIE(original);  
    }  
  
    protected String encodeForGecko(String original) {  
        try {  
            return new BCodec().encode(original);  
        }  
    }  
}
```

```
        } catch (EncodingException e) {
            return original;
        }
    }

    protected String encodeForIE(String original) {
        try {
            return URLEncoder.encode(original,
                AbstractDownloadObject.DEFAULT_CHARSET);
        } catch (UnsupportedEncodingException e) {
            return original;
        }
    }
}
```

- `DownloadAction` を継承してファイルダウンロードを行う場合 `getStreamInfo()` の実装部分で `DownloadBLogicAction` のクロスブラウザ対応と同様の対応を行う。

■ 関連機能

- 『WH-01 ビジネスロジック実行機能』
- 『WD-02 ファイルアップロード機能』

■ 使用例

なし。

■ 備考

なし。

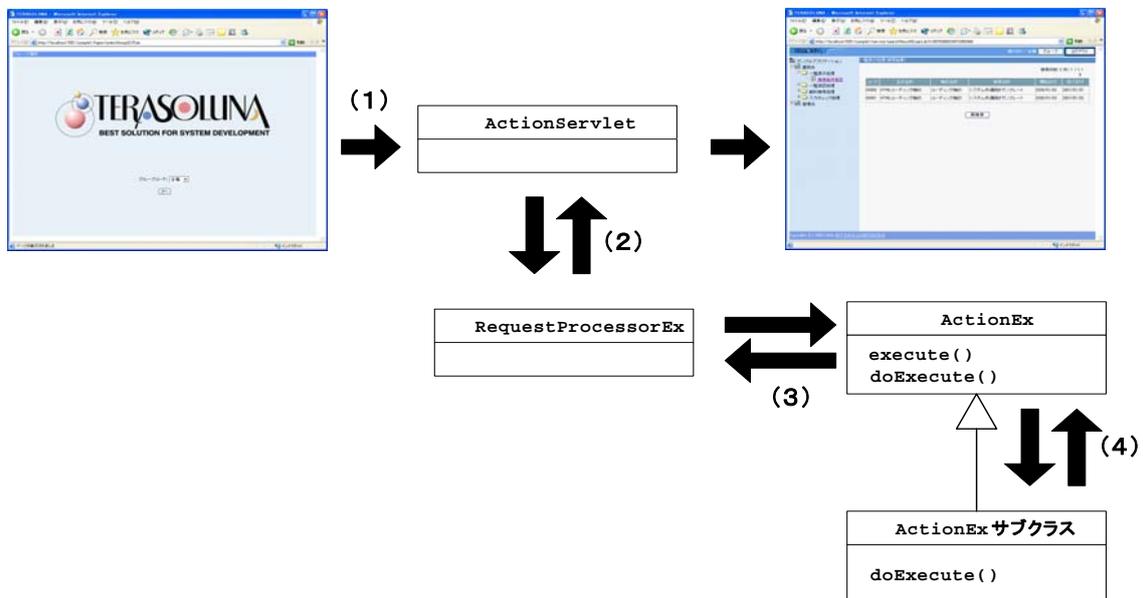
WE-01 アクション拡張機能

■ 概要

◆ 機能概要

- TERASOLUNA Server Framework for Java (Web 版)の全てのアクションクラスの基底クラスを提供する。
- 以下の機能を提供する。
 - トランザクショントークンチェックを起動する。
 - ビジネスロジックの実行後等で保存先がセッションに指定されているエラー情報、メッセージ情報をセッションスコープに設定する。
- ※ 業務アプリケーションで新規にアクションを実装する場合も、ActionEx を継承することを推奨する。
- ※ トランザクショントークンチェックとは、サブミット2度押しや、ブラウザの「戻る」ボタンを使った重複サブミットを防ぐ機能である。
- ※ Action クラスは DI コンテナで管理し、原則として scope="prototype"で使用することを推奨する。

◆ 概念図



◆ 解説

- (1) ブラウザからリクエストがサーバに送信される。
- (2) ActionServlet から RequestProcessorEx が呼び出される。
- (3) RequestProcessorEx#processActionPerform()により、ActionEx#execute()が実行される。
- (4) トランザクショントークンチェック、拡張アクションフォームの値変更フラグを false に設定し、ActionEx サブクラスの doExecute()メソッドを呼び出す。

■ 使用方法

◆ コーディングポイント

- トランザクショントークンチェック
 - トランザクショントークンチェックを行うためには、以下の設定を行う。
 - ✧ アクションクラスの Bean 定義で<property>要素の"tokenCheck"に対し、"true"を明示する必要がある。
 - ✧ struts-config.xml の<forward>要素（<global-forwards>内でも可）で"txtoken-error"という名前でトークンエラー時のパスを指定する。
 - ActionEx を使用した場合、saveToken()によって自動的にトークンが保存されるが、保存しない場合には以下の設定を行う。
 - ✧ アクションクラスの Bean 定義で<property>要素の"saveToken"に対し、"false"を明示する必要がある。

➤ Struts 設定ファイル (struts-config.xml)

```
<struts-config>
.....
<action-mappings type="jp.terasoluna.fk.web.struts.action.ActionMappingEx">
<!-- action設定 -->
<action path="/logoff"
        name="logonSampleForm"
        scope="session"
        parameter="/logoffForward.do">
    <forward name="txtoken-error" module="/sub" path="/doubleRegistError.do" />
</action>
<action path="/logoffForward" parameter="/logoff.jsp" />
</action-mappings>
.....
</struts-config>
```

拡張アクションマッピングを指定する。

トランザクショントークンチェックに失敗した場合、
論理フォワード名 "txtoken-error" に遷移する。

同名で指定する。
任意の名前でよい。

➤ Bean 定義ファイル

```
<bean name="/logoff" scope="prototype">
    class="jp.terasoluna.fw.web.struts.actions.LogoffAction">
<property name="tokenCheck">
    <value>true</value>
</property>
<property name="saveToken">
    <value>>false</value>
</property>
</bean>
<bean name="/logoffForward" scope="prototype">
    class="jp.terasoluna.fw.web.struts.actions.ForwardAction" />
```

tokenCheck プロパティに true を指定することにより、
トランザクショントークンチェックが行なわれる。デ
フォルト値は false。

saveToken プロパティに false を指定することにより、トランザクシ
ョントークンの保存が行なわれない。デフォルト値は true。※ここ
では、後に続くアクション (/logoffForward) でトークンが保存さ
れるため、あえて false に設定している。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.struts.actions.ActionEx	TERASOLUNA Server Framework for Java (Web 版)が提供する、拡張アクションクラスである。
2	jp.terasoluna.fw.web.struts.action.ActionMappingEx	拡張アクションマッピングクラスである。
3	jp.terasoluna.fw.web.struts.actions.AbstractBLogicAction	ビジネスロジックを起動する抽象アクションクラスである。 ビジネスロジックの入出力情報とアクションフォーム、セッションとの情報をやり取りしている。詳細は、『WH-01 ビジネスロジック実行機能』を参照のこと。
4	jp.terasoluna.fw.web.struts.actions.DispatchAction	リクエストパラメータの設定値に応じて遷移先を決定するアクションである。詳細は、『WE-02 標準ディスパッチャ機能』を参照のこと。
5	jp.terasoluna.fw.web.struts.actions.ForwardAction	Struts 設定ファイル<action>要素の property 属性が”parameter”であるパスに遷移するアクションである。 詳細は、『WE-03 フォワード機能』を参照のこと。
6	jp.terasoluna.fw.web.struts.actions.ReloadCodeListAction	サーブレットコンテキストに格納されているコードリストをリロードするアクションである。詳細は『WE-04 コードリストリロード機能』を参照のこと。
7	jp.terasoluna.fw.web.struts.actions.MakeSessionDirectoryAction	ユーザ固有のディレクトリを作成するアクションである。詳細は『WE-05 セッションディレクトリ作成機能』を参照のこと。
8	jp.terasoluna.fw.web.struts.actions.ClearSessionAction	Bean 定義ファイルで定義されたキーをセッション上から削除するアクションクラスである。詳細は『WE-06 セッションクリア機能』を参照のこと。
9	jp.terasoluna.fw.web.struts.actions.LogoffAction	ログオフ時にセッションを破棄するアクションである。詳細は『WE-07 ログオフ機能』を参照のこと。

◆ 拡張ポイント

TERASOLUNA Server Framework for Java (Web 版)の拡張、及び業務で新規にアクションクラスを実装する場合は、ActionEx を継承したクラスを作成すること。

■ 関連機能

- 『WE-02 標準ディスパッチャ機能』
- 『WE-03 フォワード機能』
- 『WE-04 コードリストリロード機能』
- 『WE-05 セッションディレクトリ作成機能』
- 『WE-06 セッションクリア機能』
- 『WE-07 ログオフ機能』
- 『WH-01 ビジネスロジック実行機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC16 アクション拡張」
 - ◇ /webapps/actionex/*
 - ◇ /webapps/WEB-INF/actionex/*
 - ◇ jp.terasoluna.thin.functionsample.actionex.*

■ 備考

- なし

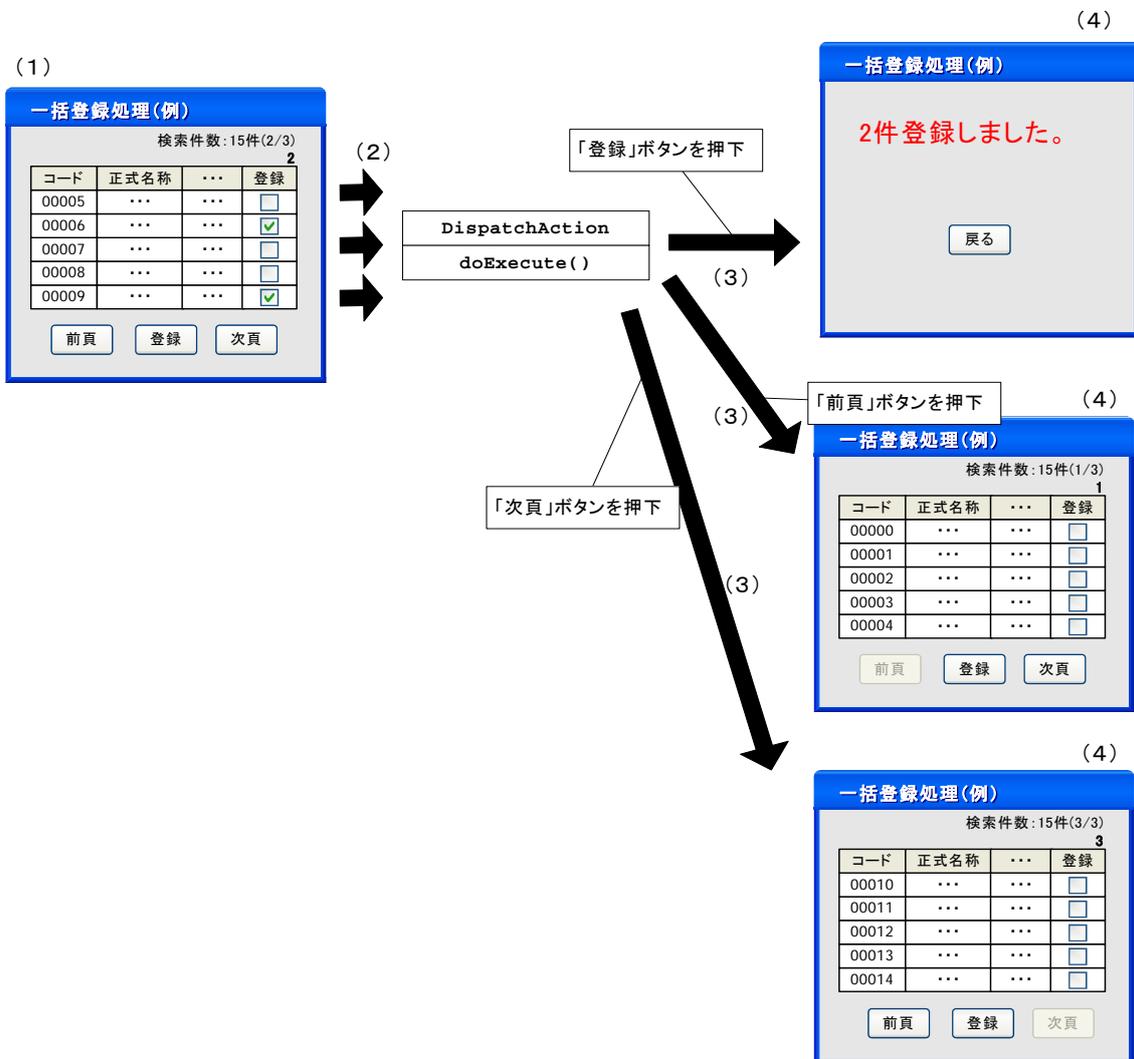
WE-02 標準ディスパッチャ機能

■ 概要

◆ 機能概要

- Terasoluna Server Framework for Java (Web 版)で提供するディスパッチャ機能として、以下の2つを提供する。
 - 押下したサブミットボタンに応じて呼び出すアクションを振り分ける処理を、JSP と Struts 設定ファイル (struts-config.xml) と Bean 定義ファイルの記述だけで対処できるようにし、個別のアクション生成を不要にする。
 - また、サブミットボタンの数が1つの画面でも、リクエストパラメータの設定によって呼び出すアクションを振り分けることができる。

◆ 概念図



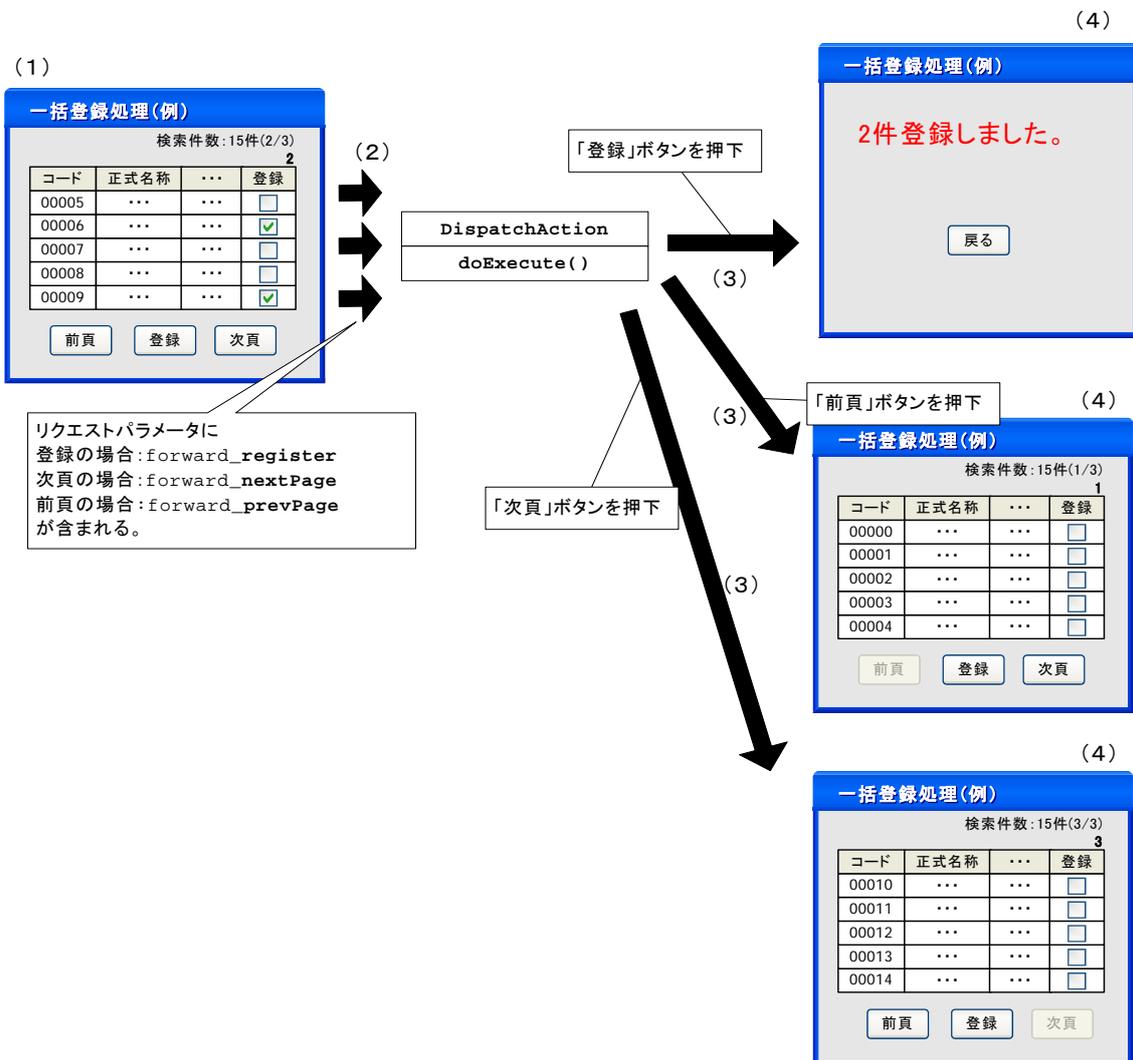
◆ 解説

- (1) 1つのJSPに対し、複数のサブミットボタンを用意する。
- (2) リクエストパラメータに”forward_XXXX”のような値を設定して送信する。
- (3) 送信されたリクエストパラメータ値から論理フォワード名を生成する。
- (4) 論理フォワード名に従って、次画面遷移を行う。

■ 使用方法

◆ コーディングポイント

- 標準ディスパッチャ機能は、下記の2通りの使用が可能となっている。
- (1) 複数のサブミットボタンにより、遷移先を振り分ける方法を示す。
本機能を使う場合、Struts 設定ファイル (struts-config.xml) と Bean 定義ファイルの設定と、JSP の編集が必要となる。
これにより「登録」ボタンが押下された時、forward_register というキーがリクエストパラメータで指定され、register を論理フォワード名として次に遷移する。



➤ Struts 設定ファイル (struts-config.xml)

```
<struts-config>
.....
<action-mappings type="jp.terasoluna.fw.web.struts.action.ActionMappingEx">
.....
<action path="/fileUploadConfirmDSP" name="_fileForm">
  <forward name="back" path="/initFileUploadAction.do"/>
  <forward name="default" path="/fileUploadConfirmSCR.do"/>
  <forward name="regist" path="/fileRegistBLogic.do"/>
</action>
<action-mappings>
.....
</struts-config>
```

アクションパス

リクエストパラメータが送られてきていない場合に
そなえ、必ず「default」という遷移先を定義する。

リクエストパラメータで送信される「forward_」を取り
除いた値ごとに、アクションフォワードを定義する。

➤ Bean 定義ファイル

```
<bean name="/sample/selectDispatch" scope="prototype" class=
  "jp.terasoluna.fw.web.struts.actions.DispatchAction">
</bean>
```

DispatchAction を指定する。

➤ JSP ファイル

```
<html:html>
.....
<html:form action="/fileUploadConfirmDSP">
.....
  <html:submit property="forward_back" value=" 戻る " />
  <html:submit property="forward_regist" value=" 登録 " />
.....
</html:form>
.....
</html:html>
```

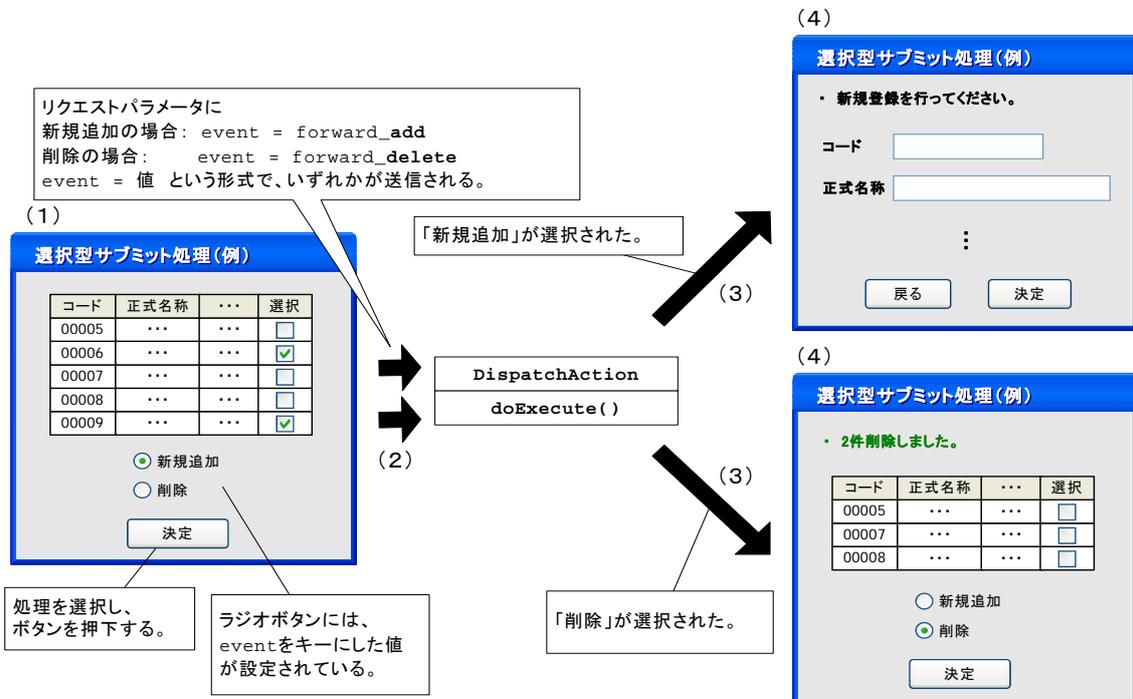
Struts 設定ファイルで定義されたアクションパスを指定する。

“forward_XXXX”という形で送信するリクエストパラメータを
指定する。

DispatchAction#doExecute()によって、XXXX の部分が
論理フォワード名として採用される。

(2) ラジオボタンにより、遷移先条件を設定する例を示す。

ここでは、フォームの”event”フィールドが既に使用されている場合を想定し、遷移先を表すリクエストパラメータのキーを、”event”から”dispatchName”に変更している。



➤ Struts 設定ファイル (struts-config.xml)

```

<struts-config>
  .....
  <action-mappings type=" jp.terasoluna.fw.web.struts.action.ActionMappingEx">
    ...
    <action path="/fileUploadConfirmDSP" name="_fileForm">
      <forward name="back" path="/initFileUploadAction.do"/>
      <forward name="default" path="/fileUploadConfirmSCR.do"/>
      <forward name="regist" path="/fileRegistBLogic.do"/>
    </action>
  </action-mappings>
  .....
</struts-config>

```

アクションパス

リクエストパラメータが送られてきていない場合に
そなえ、必ず「default」という遷移先を定義する。

リクエストパラメータで送信される”forward_”を取り
除いた値ごとに、アクションフォワードを定義する。

➤ Bean 定義ファイル

```
<bean name="/sample/selectDispatch" scope="prototype"
  class="jp.terasoluna.fw.web.struts.actions.DispatchAction">
  <property name="event">
    <value>dispatchName</value>
  </property>
</bean>
```

DispatchAction を指定する。

振り分け先を指定するリクエストパラメータのキーを、event から dispatchName に変更している。

➤ JSP ファイル

```
<html:html>
.....
<html:form action="/fileUploadConfirmDSP">
.....
  <html:radio property="dispatchName" value="forward_back"/> 戻る
  <html:radio property="dispatchName" value="forward_regist"/> 登録
.....
  <html:submit property="decide" value=" 決定 " >
</html:form>
.....
</html:html>
```

Struts 設定ファイルで定義されたアクションパスを指定する。

Struts 設定ファイルで定義されたリクエストパラメータのキーを指定する。

リクエストパラメータの値に“forward_” + 論理フォワード名と定義する。

◆ 拡張ポイント

キャンセルボタン押下時の遷移先を決定する場合、このクラスを継承して cancelled() メソッドをオーバーライド実装する。

下記は、フォワード先の論理名“cancelled”を元に、キャンセル時の遷移先を決定する実装例である。

ここでは、/fileUploadConfirmDSP.do というアクション処理で、キャンセルボタンが押下された場合に/backward.do に遷移する。

➤ DispatchAction 継承クラス (DispatchActionEx)

```
public class DispatchActionEx extends DispatchAction {
.....
  protected ActionForward cancelled(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response) {
    // アクションマッピングから、キャンセル時のフォワード先を取得する。
    ActionForward forward = mapping.findForward("cancelled");
    return forward;
  }
.....
}
```

DispatchAction を継承する。

Struts 設定ファイルのアクションフォワード定義から、“cancelled”で定義された遷移先を検索し、ActionForward インスタンスを返却する。

➤ Struts 設定ファイル (struts-config.xml)

```
<action-mappings
  type="jp.terasoluna.fw.web.struts.action.ActionMappingEx">
  .....
  <action path="/fileUploadConfirmDSP" name="_fileForm">
    <forward name="back" path="/initFileUploadAction.do"/>
    <forward name="default" path="/fileUploadConfirmSCR.do"/>
    <forward name="regist" path="/fileRegistBLogic.do"/>
    <forward name="cancelled" path="/backward.do"/>
  </action>
</action-mappings>
```

キャンセルボタンが押された場合の遷移先を"cancelled"というアクションフォワード定義で指定している。

➤ Bean 定義ファイル

```
<bean name="/fileUploadConfirmDSP" scope="prototype"
  class="jp.terasoluna.sample.action.DispatchActionEx">
</bean>
```

上記で作成した、DispatchAction の継承クラスを指定する。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.struts.actions.DispatchAction	次に遷移するパスを振り分けるディスパッチャクラスである。 リクエストパラメータに event、または forward_XXX が指定されている場合、これらの値を用いて振り分け先の論理フォワード名が取得される。

■ 関連機能

- なし

■ 使用例

- Terasoluna Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC17 標準ディスパッチャ」
 - ◇ /webapps/dispatch/*
 - ◇ /webapps/WEB-INF/dispatch/*
 - ◇ jp.terasoluna.thin.functionsample.dispatch.*

■ 備考

- フォワード先を決めるリクエストパラメータが送られてきていない場合
一部のブラウザには HTML フォームのテキストフィールドでエンターキーを押下すると、サブミットボタンの name 属性、value 属性がリクエストパラメータに含まれずにサブミットするものがある。
この場合 DispatchAction の「ボタンで遷移する場合」では、遷移先が送られてきていないため、「default」というフォワード名で定義された遷移先に遷移する仕様になっている。

default 遷移先を Struts 設定ファイルに定義していない場合、空白画面が表示されるので、かならず定義しておくこと。

- フォワード名が Struts 設定ファイルに定義されていない場合
Struts 設定ファイルに以下のように遷移先が定義されているとき、
 1. 設定ファイルの記述ミス
 2. リクエストパラメータの改ざん等により、定義されていないフォワード名がリクエストパラメータとして送信された場合 (event=forward_notdefined または forward_notdefined=yyyy が送信された場合)、default 遷移先に遷移する。

➤ Struts 設定ファイル (struts-config.xml)

```
<action-mappings
  type="jp.terasoluna.fw.web.struts.action.ActionMappingEx">
  .....
  <action path="/registerDSP" name="_registerForm">
    <forward name="regist" path="/registerBLogic.do"/>
    <forward name="default" path="/backward.do"/>
  </action>
</action-mappings>
```

notdefined というフォワード名は定義されていないので、default 遷移先に遷移する。

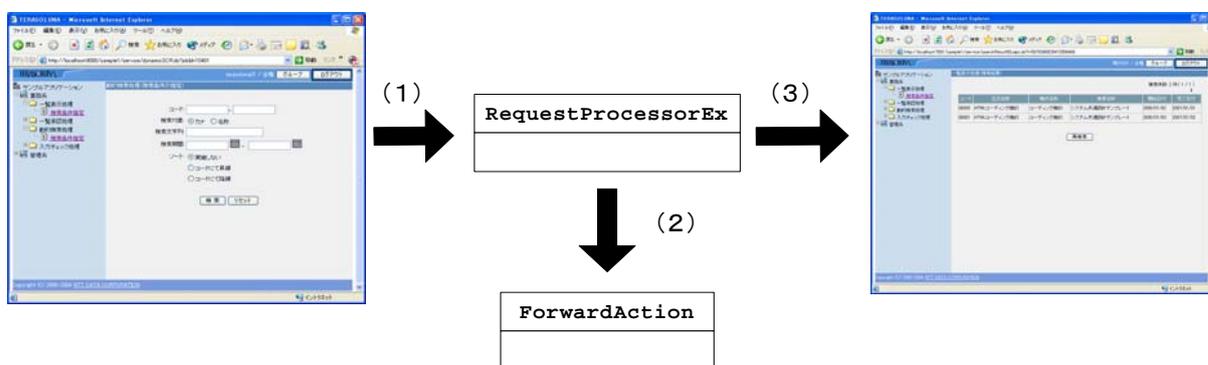
WE-03 フォワード機能

■ 概要

◆ 機能概要

- TERASOLUNA Server Framework for Java (Web 版)では、原則としてブラウザから拡張子が”jsp”である URI へのアドレス直接指定によるアクセスを禁止している。そのため遷移ログを出力し、JSP を表示するためのアクションを提供する。
- ※ JSP へのアドレス直接指定によるアクセス禁止機能の詳細は、『WA-03 拡張子直接指定禁止機能』を参照のこと。
- 遷移先は、Struts が提供している ForwardAction と同様に struts-config.xml で <action>要素の parameter 属性に指定する方法と、<forward>要素で指定する方法がある。リダイレクトやモジュール間の遷移を行ないたい場合は、<forward>要素による指定を行うこと。

◆ 概念図



◆ 解説

- (1) ブラウザからリクエストを送信する。
- (2) RequestProcessorEx が、ForwardAction を起動する。ForwardAction クラスは、ビジネスロジックの起動などの処理は行わず、ログの出力と継承元の ActionEx クラスの機能を用い、Struts 設定ファイル (struts-config.xml) の<action>要素で parameter 属性をパスに指定した ActionForward を返却する。parameter 属性が設定されていない場合は、論理フォワード名 success の ActionForward を返却する。
ActionEx についての詳細は、『WE-01 アクション拡張機能』を参照のこと。
- (3) ActionForward のパスにフォワードする。フォワード先が設定されていない場合、SC_NOT_FOUND (404) エラーを返す。

■ 使用方法

◆ コーディングポイント

- フォワード機能の実装は、Struts 設定ファイル (struts-config.xml) と Bean 定義ファイルを設定することで利用可能となる。

- <action>要素の parameter 属性で遷移先を指定する場合は、以下のようにコンテキストルートからの絶対パスを設定する。

➤ Struts 設定ファイル (struts-config.xml)

```
<action path="/foo" parameter="/sub1/foo.jsp">
</action>
```

parameter 属性に指定した先にフォワードする。

➤ Bean 定義ファイル

```
<bean name="/foo" scope="prototype"
class="jp.terasoluna.fw.web.struts.actions.ForwardAction">
</bean>
```

同名で指定する。
任意の名前でよい。

ForwardAction を指定する。

- <forward>要素で遷移先を指定する場合は、以下のように設定する。上記 parameter 要素に設定するモジュール相対パスでは指定できないモジュールを跨いだパスへの遷移や、リダイレクトを行なえる。

<action>要素の parameter 属性が設定されていた場合、parameter 属性を遷移先として優先するので注意すること。

➤ Struts 設定ファイル (struts-config.xml)

```
<action path="/foo">
<forward name="success" path="/foo.jsp" module="/sub1"/>
</action>
```

論理フォワード名 success 固定で遷移先を指定する。

同名で指定する。
任意の名前でよい。

➤ Bean 定義ファイル

```
<bean name="/foo" scope="prototype"
class="jp.terasoluna.fw.web.struts.actions.ForwardAction">
</bean>
```

ForwardAction を指定する。

- <action>要素の `parameter` 属性で遷移先を指定する場合のパスの指定方法を変更する場合は、プロパティファイルに以下のように設定する。
`forwardAction.contextRelative` が `true` の場合は、コンテキストルートからの絶対パスで指定する。`false` の場合はモジュールパスからの相対パスで指定する。このオプションを指定しなかった場合のデフォルトは `false` で動作する。

➤ `system.properties` ファイル

```
# ForwardActionのcontextRelative設定
# true : Struts準拠
# false : Terasoluna 1.x系 2.0.0.x系 互換モード
forwardAction.contextRelative=false
```

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.struts.actions.ForwardAction	次の遷移先に遷移させるアクションクラスである。
2	jp.terasoluna.fw.web.struts.actions.ActionEx	ActionEx#execute()で行われている処理は ForwardAction で継承される。

◆ 拡張ポイント

なし

■ 関連機能

- 『WA-03 拡張子直接指定禁止機能』
- 『WE-01 アクション拡張機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC18 フォワード」
 - ◇ /webapps/forward/*
 - ◇ /webapps/WEB-INF/forward/*
 - ◇ jp.terasoluna.thin.functionsample.actionex.web.action.ActionExAction.java 等
- TERASOLUNA Server Framework for Java (Web 版) チュートリアル
 - 「2.3 画面遷移」
 - /webapps/WEB-INF/moduleContext.xml

■ 備考

- なし

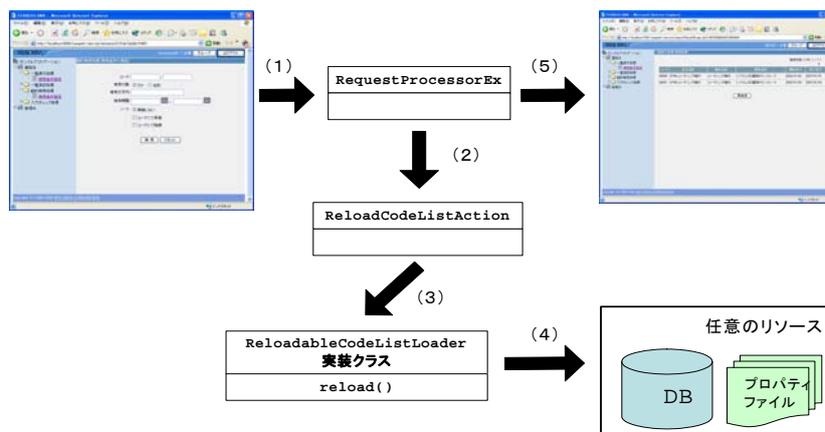
WE-04 コードリスト再読み込み機能

■ 概要

◆ 機能概要

- ReloadCodeListAction 実行時にコードリストデータの再読み込みを行う。この機能を用いることにより、コードリストデータの更新を行うことが可能となる。
- コードリストデータはDB またはプロパティファイルから取得可能である。但し、Web アプリケーションが war ファイル、ear ファイルといったアーカイブ内に収められている場合、プロパティファイルの編集が不可能である為、実質プロパティファイルによるコードリスト定義のリロードは不可能である。
ReloadableCodeListLoader インタフェース実装クラスについては『WB-05 コードリスト機能』を参照のこと。

◆ 概念図



◆ 解説

- (1) ブラウザからリクエストを送信する。
- (2) `RequestProcessorEx` が、`ReloadCodeListAction` を起動する。`ReloadCodeListAction` は (3) の処理を行った後、ログの出力と継承元の `ActionEx` クラスの機能を用い、Struts 設定ファイル (`struts-config.xml`) の `<action>` 要素で `parameter` 属性をパスに指定した `ActionForward` を返却する。
`ActionEx` についての詳細は、『WE-01 アクション拡張機能』を参照のこと。
- (3) `ReloadCodeListAction` クラスは Spring 設定ファイルの `codeListLoader` プロパティに設定された `ReloadableCodeListLoader` の実装クラスを取得する。
- (4) (3) で取得した `ReloadableCodeListLoader` の `reload()` メソッドが呼び出され、コードリストの再読み込みを行う。
- (5) `ActionForward` のパスにフォワードする。フォワード先が設定されていない場合、`SC_NOT_FOUND` (404) エラーを返す。

■ 使用方法

◆ コーディングポイント

- コードリストデータリロード機能の実装は、Struts 設定ファイル（struts-config.xml）と Bean 定義ファイルを設定することで利用可能となる。

➤ Struts 設定ファイル（struts-config.xml）

```
<action path="/reload" parameter="/result.jsp" />
```

parameter 属性に指定した先にフォワードする。

同名で指定する。
任意の名前でよい。

➤ Bean 定義ファイル

```
<bean name="/reload" scope="prototype"
      class="jp.terasoluna.fw.web.struts.actions.ReloadCodeListAction">
```

ReloadCodeListAction を指定する。

```
<property name="codeListLoader"
```

```
<ref bean="codeList" />
```

```
</property>
```

```
</bean>
```

更新対象の ReloadableCodeListLoader を指定する。

```
<bean id="codeList"
      class="jp.terasoluna.fw.web.codelist.DBCodeListLoader"
```

```
init-method="load">
```

```
<property name="dataSource">
```

```
<ref bean="TerasolunaDataSource" />
```

```
</property>
```

```
<property name="sql">
```

```
<value>SELECT KEY, LABEL FROM CODE_LIST</value>
```

```
</property>
```

```
</bean>
```

TERASOLUNA が提供する
ReloadableCodeListLoader 実装クラス

dataSource プロパティにはコードリスト情報を読み込む対象のデータソースを指定。sql にはコードリストを取得する SQL 文を設定する。この例では KEY がコードリストの id、LABEL がコードリストの値となる。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.struts.actions.ReloadCodeListAction	次の遷移先に遷移させるアクションクラスである。
2	jp.terasoluna.fw.web.struts.actions.ActionEx	ActionEx#execute()で行われている処理は ReloadCodeListAction で継承される。
3	jp.terasoluna.fw.web.codelist.ReloadableCodeListLoader	再読み込みを行うコードリスト機能のインタフェースである。詳細は『WB-05 コードリスト機能』を参照のこと。
4	jp.terasoluna.fw.web.codelist.DBCodeListLoader	コードリスト情報の初期化を、データベースを用いて行う ReloadableCodeListLoader 実装クラスである。詳細は『WB-05 コードリスト機能』を参照のこと。

◆ 拡張ポイント

上記『実装ポイント』の Spring 設定ファイル (struts-config.xml) で示した、ReloadableCodeListLoader インタフェース実装クラスを拡張することにより、コードリスト読み込み機能の拡張を行うことが可能である。

ReloadableCodeListLoader インタフェース実装クラスについては『WB-05 コードリスト機能』を参照のこと。

■ 関連機能

- 『WB-05 コードリスト機能』
- 『WE-01 アクション拡張機能』

■ 使用例

- TERASOLUNA 機能網羅サンプル Web ブラウザ対応版
 - 「UC13 コードリスト」
 - ◇ /webapps/codelist/*
 - ◇ /webapps/WEB-INF/codelist/*
 - ◇ jp.terasoluna.thin.functionsample.codelist.*

■ 備考

- なし

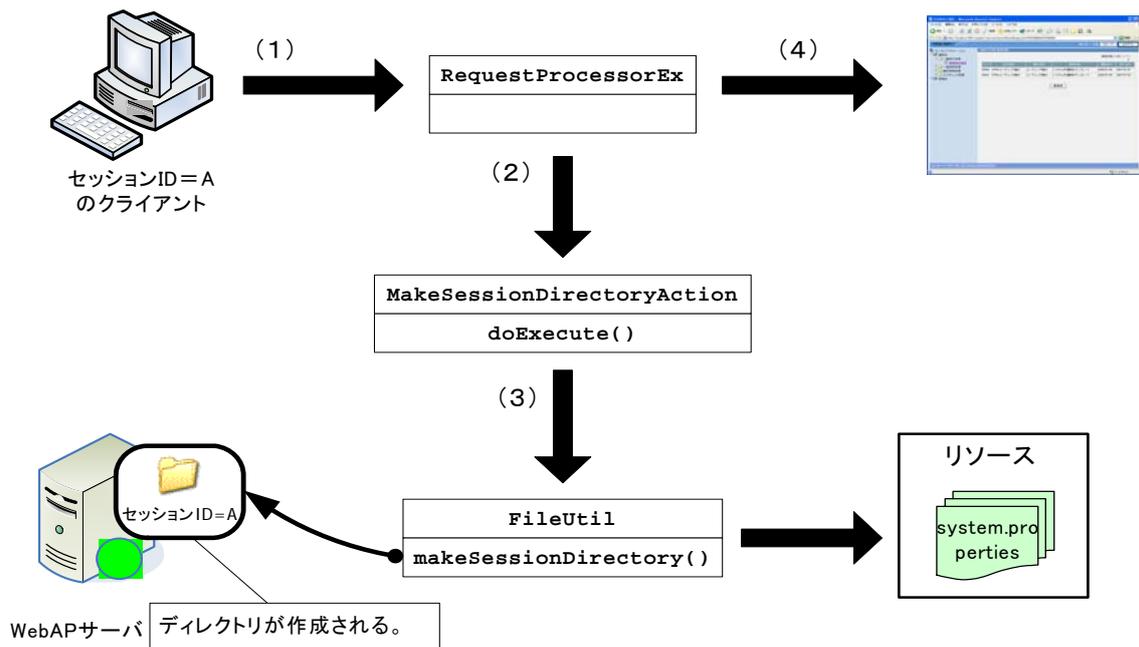
WE-05 セッションディレクトリ作成機能

■ 概要

◆ 機能概要

- サーバサイドで生成された PDF ファイルなどを格納するための一時ディレクトリ（以降、セッションディレクトリ）をログオンユーザ毎に作成する。
- セッションが無効化されるとセッションリスナのコールバックが行われ、セッションディレクトリが削除される。詳細は『WD-01 セッションディレクトリ機能』を参考のこと。

◆ 概念図



◆ 解説

- (1) ログオン時など、ユーザからリクエストが送信される。
- (2) RequestProcessorEx が、MakeSessionDirectoryAction を起動する。
MakeSessionDirectoryAction クラスは(3)の処理を行った後、ログの出力と継承元の ActionEx クラスの機能を用い、Struts 設定ファイル (struts-config.xml) の<action>要素で parameter 属性をパスに指定した ActionForward を返却する。
ActionEx についての詳細は、『WE-01 アクション拡張機能』を参照のこと。
- (3) セッションディレクトリを作成する。
 - FileUtil#makeSessionDirectory()の処理を行う。
FileUtil についての詳細は、『CD-01 ユーティリティ機能』を参照のこと。
- (4) ActionForward のパスにフォワードする。フォワード先が設定されていない場合、SC_NOT_FOUND (404) エラーを返す。

■ 使用方法

◆ コーディングポイント

- セッションディレクトリが配置されるベースとなるディレクトリを決定し、システム設定プロパティファイル (system.properties) に設定する。
 - システム設定プロパティファイル (system.properties)

```
session.dir.base=/tmp/sessions
```

/tmp/session 配下に、セッションディレクトリが格納される。

- TERASOLUNA が提供するセッションディレクトリ作成クラスのアクションマッピング定義を設定する。
 - Struts 設定ファイル (struts-config.xml)

```
<action path="/makeSessionDir"  
  scope="session"  
  parameter="/foo.jsp">  
</action>
```

同名で指定する。
任意の名前でよい。

- Bean 定義ファイル

```
<bean name="/makeSessionDir" scope="prototype" class=  
  "jp.terasoluna.fw.web.struts.actions.MakeSessionDirectoryAction">  
</bean>
```

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.util.FileUtil	ファイル・ディレクトリの生成、削除、取得を行うユーティリティクラスである。詳細は『CD-01 ユーティリティ機能』を参照のこと。
2	jp.terasoluna.fw.web.struts.actions.MakeSessionDirectoryAction	セッションディレクトリをログオンユーザ毎に作成する。
3	jp.terasoluna.fw.web.struts.actions.ActionEx	ActionEx#execute()で行われている処理は MakeSessionDirectoryAction で継承される。

◆ 拡張ポイント

なし

■ 関連機能

- 『CD-01 ユーティリティ機能』
- 『WD-01 セッションディレクトリ機能』

■ 使用例

- Terasoluna 機能網羅サンプル Web ブラウザ対応版
 - 「UC15 セッションディレクトリ」
 - ◇ /webapps/sessiondir/*
 - ◇ /webapps/WEB-INF/sessiondir/*
 - ◇ jp.terasoluna.thin.functionsample.sessiondir.*

■ 備考

- なし

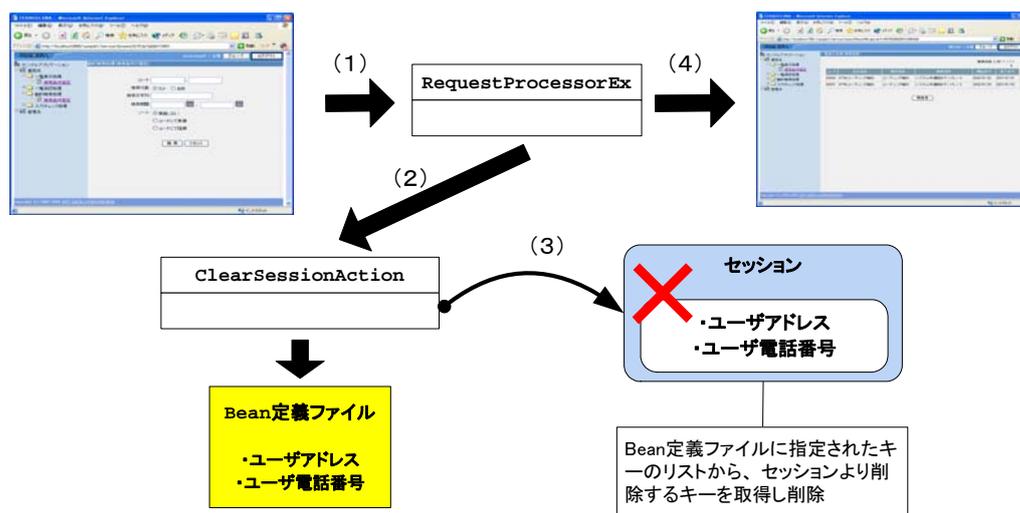
WE-06 セッションクリア機能

■ 概要

◆ 機能概要

- セッションから指定されたプロパティを削除する。
 - Bean 定義ファイルに指定されたキーのリストから、セッションより削除するキーを取得し削除を行う。
- 現在のセッションそのものを破棄する場合は、『WE-07 ログオフ機能』を参照のこと。

◆ 概念図



◆ 解説

- (1) ブラウザからリクエストを送信する。
- (2) RequestProcessorEx が、ClearSessionAction を起動する。ClearSessionAction クラスは (3)の処理を行った後、ログの出力と継承元の ActionEx クラスの機能を用い、Struts 設定ファイル (struts-config.xml) の<action>要素で parameter 属性をパスに指定した ActionForward を返却する。
ActionEx についての詳細は、『WE-01 アクション拡張機能』を参照のこと。
- (3) Bean 定義ファイルに指定されたキーのリストから、セッションより削除するキーを取得し削除を行う。
- (4) ActionForward のパスにフォワードする。フォワード先が設定されていない場合、SC_NOT_FOUND (404) エラーを返す。

■ 使用方法

◆ コーディングポイント

- TERASOLUNA Server Framework for Java (Web 版)が提供する ClearSessionAction のアクションマッピング定義を設定する。

➤ Struts 設定ファイル (struts-config.xml)

```
<action path="/clearSession"
  scope="session"
  parameter="/foo.jsp">
</action>
```

同名で指定する。
任意の名前でよい。

- セッションから削除するキーの値を、Bean 定義ファイルの clearSessionKeys プロパティで設定する。

➤ Bean 定義ファイル

```
<bean name="/clearSession" scope="prototype"
  class="jp.terasoluna.fw.web.struts.actions.ClearSessionAction">
  <property name="clearSessionKeys">
    <list>
      <value>userAddress</value>
      <value>userPhoneNo</value>
      <value>sampleSession</value>
    </list>
  </property>
</bean>
```

セッションから削除対象のキーの値
を list プロパティで定義する。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.struts.actions.ClearSessionAction	セッションから指定されたプロパティを削除する。
2	jp.terasoluna.fw.web.struts.actions.ActionEx	ActionEx#execute()で行われている処理は ClearSessionAction で継承される。

◆ 拡張ポイント

なし。

■ 関連機能

- 『WE-01 アクション拡張機能』
- 『WE-07 ログオフ機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC19 セッションクリア」
 - ◇ /webapps/clearsession/*
 - ◇ /webapps/WEB-INF/clearsession/*
 - ◇ jp.terasoluna.thin.functionsample.clearsession.*

■ 備考

- なし

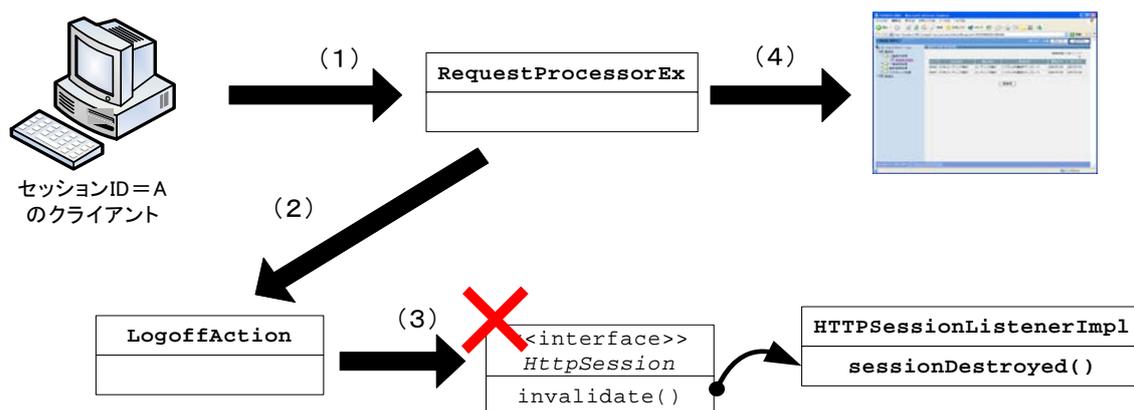
WE-07 ログオフ機能

■ 概要

◆ 機能概要

- サーバサイドで生成された PDF ファイルなどを格納するための一時ディレクトリ（以降、セッションディレクトリ）をログオンユーザ毎に作成する。
- セッションが無効化されるとセッションリスナのコールバックが行われ、セッションディレクトリが削除される。詳細は『WD-01 セッションディレクトリ機能』を参考のこと。

◆ 概念図



◆ 解説

- (1) ログオフ時など、ユーザからリクエストが送信される。
- (2) RequestProcessorEx が、LogoffAction を起動する。LogoffAction クラスは(3)の処理を行った後、ログの出力と継承元の ActionEx クラスの機能を用い、Struts 設定ファイル（struts-config.xml）の<action>要素で parameter 属性をパスに指定した ActionForward を返却する。
ActionEx についての詳細は、『WE-01 アクション拡張機能』を参照のこと。
- (3) セッションの無効化処理を行う。
セッションが無効化されると、HttpSessionListenerImpl#sessionDestroyed()メソッドのコールバックが行われる。
- (4) ActionForward のパスにフォワードする。フォワード先が設定されていない場合、SC_NOT_FOUND (404) エラーを返す。

■ 使用方法

◆ コーディングポイント

- Terasoluna Server Framework for Java (Web 版)が提供する LogoffAction のアクションマッピング定義を設定する。

➤ Struts 設定ファイル (struts-config.xml)

```
<action path="/logoff"
  scope="session"
  parameter="/foo.jsp">
</action>
```

同名で指定する。
任意の名前でよい。

➤ Bean 定義ファイル

```
<bean name="/logoff" scope="prototype"
  class="jp.terasoluna.fw.web.struts.actions.LogoffAction">
</bean>
```

TERASOLUNA が提供
するログオフクラス。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.struts.actions.LogoffAction	ログオフ処理を実行するアクションクラスである。
2	jp.terasoluna.fw.web.struts.actions.ActionEx	ActionEx#execute()で行われている処理は LogoffAction で継承される。
3	jp.terasoluna.fw.web.HttpSessionListenerImpl	セッションの生成、無効化時の動作を定義する HttpSessionListener インタフェースを実装したクラスである。

◆ 拡張ポイント

なし。

■ 関連機能

- 『WD-01 セッションディレクトリ機能』
- 『WE-01 アクション拡張機能』

■ 使用例

- Terasoluna Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC00 共通機能」
 - ◇ /webapps/*
 - ◇ /webapps/WEB-INF/*
 - ◇ jp.terasoluna.thin.functionsample.common.*

- Terasoluna Server Framework for Java (Web 版) チュートリアル
 - 「2.4 ログオン」
 - /webapps/WEB-INF/moduleContext.xml

■ 備考

- なし

WF-01 拡張入力チェック機能

■ 概要

◆ 機能概要

- commons-validator を利用し、アプリケーションで使用される典型的な入力チェックルールを提供する。

◆ 提供ルール一覧

以下は TERASOLUNA Server Framework for Java (Web 版) で提供する入力チェックルールの一覧である。

※Struts から提供されるものは除く。

ルール名	概要	サーバサイド	クライアントサイド
alphaNumericString	半角英数字文字列チェック	○	○
hankakuKanaString	半角カナ文字列チェック	○	○
hankakuString	半角文字列チェック	○	○
zenkakuString	全角文字列チェック	○	○
zenkakuKanaString	全角カナ文字列チェック	○	○
capAlphaNumericString	大文字英数字文字列チェック	○	○
number	数値チェック	○	○
numericString	数字文字列チェック	○	○
prohibited	禁止文字列チェック	○	×
stringLength	文字列長チェック	○	○
byteLength	byte 列長チェック	○	×
byteRange	byte 列長範囲チェック	○	×
dateRange	date 型範囲チェック	○	○
multiFieldCheck	複数フィールド関連チェック	○	×

■ 使用方法

◆ コーディングポイント

拡張入力チェック機能の初期化方法を以下に示す。各ルールの使用方法は次章以降を参照のこと。

- Struts 設定ファイル

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
  <set-property
    property="pathnames"
    value="/WEB-INF/validator-rules.xml,/WEB-INF/validator-rules-
ex.xml,/WEB-INF/validation.xml"/>
  <set-property
    property="stopOnFirstError" value="false"/>
</plug-in>
```

拡張入力チェック機能のルールファイル。TERASOLUNA Server Framework for Java (Web 版)が提供するファイルをそのまま利用する。

■ 入力チェックルール解説

◆ alphaNumericString

入力文字列が半角の英数字のみであることをチェックする。

- 入力チェック設定ファイル (validation.xml) の例

```
.....
<form name="/validate">
  <field property="userName"
    depends="alphaNumericString">
    <arg key="label.userName" position="0" />
  </field>
</form>
.....
```

デフォルトのメッセージは「errors.alphaNumericString={0}には半角英数字で入力してください。」

◆ zenkakuString

入力文字列が全角文字のみであることをチェックする。

- 入力チェック設定ファイル (validation.xml) の例

```

.....
<form name="/validate">
  <field property="userName"
    depends="zenkakuString">
    <arg key="label.userName" position="0" />
  </field>
</form>
.....

```

デフォルトのメッセージは「errors.zenkakuString={0}には全角文字を入力してください。」

全角文字は `hankakuString` ルールの論理否定演算の結果が真であるかどうかで判定を行う。`hankakuString` ルールについては `hankakuString` ルールの説明を参照のこと。

◆ zenkakuKanaString

入力文字列が全角カナ文字のみであることをチェックする。

- 入力チェック設定ファイル (validation.xml) の例

```

.....
<form name="/validate">
  <field property="userName"
    depends="zenkakuKanaString">
    <arg key="label.userName" position="0" />
  </field>
</form>
.....

```

デフォルトのメッセージは「errors.zenkakuKanaString={0}には全角カナ文字を入力してください。」

全角カナ文字はデフォルトで以下の文字が該当する。

- アイウヴエオアイウエオカキクケコカヶガギグゲゴサシスセソザジズゼゾタチツテトダヂヅデドナニヌネノハヒフヘホバビブベボパピプペポマミムメモヤユヨャュョラリルレロワヰヱヲン

全角カナ文字に該当する文字はプロパティファイルに登録することで変更できる。

- プロパティファイル

```
validation.zenkaku.kana.list=アイウエオカキクケコサシスセソ……
```

◆ capAlphaNumericString

入力文字列が大文字の半角英数字のみであることをチェックする。

- 入力チェック設定ファイル (validation.xml) の例

```

.....
<form name="/validate">
  <field property="userName"
    depends="capAlphaNumericString">
    <arg key="label.userName" position="0" />
  </field>
</form>
.....

```

デフォルトのメッセージは「errors.capAlphaNumericString ={0} には英大文字または数字を入力してください。」

◆ number

入力文字列が数値に変換できるかどうかをチェックする。

- 入力チェック設定ファイル (validation.xml) に設定を要する<var>要素

var-name	var-value	必須性	概要
integerLength	整数部桁数	false	整数の桁数を設定する。isAccordedInteger 指定が無いときは、指定桁数以内の検証を行う。省略時、非数値を設定したときは、検証を行わない。
scale	小数部桁数	false	小数値の桁数を設定する、isAccordedScale 指定が無いときは、指定桁数以内の検証を行う。省略時、非数値を設定したときは、検証を行わない
isAccordedInteger	整数桁数一致チェック	false	true であれば、整数桁数の一致チェックが 行なわれる。省略時、true 以外の文字列が設定された時は桁数以内チェックとなる。
isAccordedScale	小数桁数一致チェック	false	true であれば、小数桁数の一致チェックが 行なわれる。省略時、true 以外の文字列が設定された時は桁数以内チェックとなる。

- 入力チェック設定ファイル (validation.xml) の例

```
.....
<form name="/validate">
  <field property="userName"
    depends="number">
    <arg key="label.userName" position="0" />
    <arg key="{var:integerLength}" position="1" resource="false"/>
    <arg key="{var:scale}" position="2" resource="false"/>
    <var>
      <var-name>integerLength</var-name>
      <var-value>3</var-value>
    </var>
    <var>
      <var-name>scale</var-name>
      <var-value>2</var-value>
    </var>
    <var>
      <var-name>isAccordedInteger</var-name>
      <var-value>>false</var-value>
    </var>
    <var>
      <var-name>isAccordedScale</var-name>
      <var-value>>true</var-value>
    </var>
  </field>
</form>
.....
```

デフォルトのメッセージは「errors.number={0}には整数部 {1}桁、小数部 {2}桁までの数値を入力してください..」

この設定例では「整数部が3桁以内で小数部が2桁」のルールで数値チェックを行う

◆ numericString

入力文字列が数字のみであることをチェックする。

- 入力チェック設定ファイル (validation.xml) の例

```
.....
<form name="/validate">
  <field property="userName"
    depends="numericString">
    <arg key="label.userName" position="0" />
  </field>
</form>
.....
```

デフォルトのメッセージは「errors.numericString={0}には数字を入力してください。」

◆ prohibited

入力文字列に入力を禁止した文字列が含まれていないことをチェックする。

- 入力チェック設定ファイル (validation.xml) に設定を要する<var>要素

var-name	var-value	必須性	概要
chars	入力禁止キャラクタ	false	入力文字列が、入力禁止キャラクタの何れかに該当した場合はエラーとする。省略時はチェックを行わない。

- 入力チェック設定ファイル (validation.xml) の例

```

.....
<form name="/validate">
  <field property="userName"
    depends="prohibited">
    <arg key="label.userName" position="0" />
    <arg key="{var:chars}" position="1" resource="false" />
    <var>
      <var-name>chars</var-name>
      <var-value>!"#$%&'(</var-value>
    </var>
  </field>
</form>
.....

```

デフォルトのメッセージは「errors.prohibited={0}に
入力禁止文字“{1}”が含まれています。」

入力された文字列に「!"#\$%&'()」が含まれていたら入力チェックエラーとする

- 備考

このルールはクライアントサイドでの入力チェックはサポートしていない。

◆ stringLength

入力文字列の桁数をチェックする。

- 入力チェック設定ファイル (validation.xml) に設定を要する<var>要素

var-name	var-value	必須性	概要
stringLength	入力文字列桁数	false	入力文字列桁数が指定された桁数と一致しない場合はエラーとする。省略時はチェックを行わない。

- 入力チェック設定ファイル (validation.xml) の例

```
.....
<form name="/validate">
  <field property="userName"
    depends="stringLength">
    <arg key="label.userName" position="0" />
    <arg key="{var:stringLength}" position="1" resource="false" />
    <var>
      <var-name>stringLength</var-name>
      <var-value>5</var-value>
    </var>
  </field>
</form>
.....
```

デフォルトのメッセージは「errors.stringLength={0}は{1}文字にしてください。」

入力された文字列の桁数が5文字であるかどうかチェックする

◆ byteLength

入力された文字列のバイト数をチェックする。

- 入力チェック設定ファイル (validation.xml) に設定を要する<var>要素

var-name	var-value	必須性	概要
byteLength	入力文字列バイト数	false	入力文字列のバイト数が指定されたバイト数と一致しない場合はエラーとする。省略時はチェックを行わない。
encoding	バイト数変換時文字コード	false	入力された文字列をバイト配列に変換する際に使用される文字コード。省略時は VM のデフォルトの文字コードが使用される。

- 入力チェック設定ファイル (validation.xml) の例

```

.....
<form name="/validate">
  <field property="userName"
    depends="byteLength">
    <arg key="label.userName" position="0" />
    <arg key="5" position="1" resource="false" />
    <var>
      <var-name>byteLength</var-name>
      <var-value>10</var-value>
    </var>
    <var>
      <var-name>encoding</var-name>
      <var-value>Windows-31J</var-value>
    </var>
  </field>
</form>
.....

```

デフォルトのメッセージは「errors.byteLength={0}は{1}文字にしてください。」

入力された文字列のバイト数が 10 バイトであるかどうかチェックする

- 備考

このルールはクライアントサイドでの入力チェックはサポートしていない。

◆ byteRange

入力文字列が指定されたバイト数範囲内であることをチェックする。

- 入力チェック設定ファイル (validation.xml) に設定を要する<var>要素

var-name	var-value	必須性	概要
maxByte	最大バイト数	false	入力文字列バイト長を検証するための最大バイト長。省略した場合は int 型の最大値。
minByte	最小バイト数	false	入力文字列バイト長を検証するための最小バイト長。省略した場合は 0。
encoding	バイト数変換時文字コード	false	入力された文字列をバイト配列に変換する際に使用される文字コード。省略時は VM のデフォルトの文字コードが使用される。

- 入力チェック設定ファイル (validation.xml) の例

```

.....
<form name="/validate">
  <field property="userName"
    depends="byteRange">
    <arg key="label.userName" position="0" />
    <arg key="{var:minByte}" position="1" resource="false" />
    <arg key="{var:maxByte}" position="2" resource="false" />
    <var>
      <var-name>maxByte</var-name>
      <var-value>16</var-value>
    </var>
    <var>
      <var-name>minByte</var-name>
      <var-value>8</var-value>
    </var>
    <var>
      <var-name>encoding</var-name>
      <var-value>Windows-31J</var-value>
    </var>
  </field>
</form>
.....

```

デフォルトのメッセージは「{0}には{1}から{2}までの範囲で入力してください。」

入力された文字列のバイト数が 8 バイト以上、16 バイト以下であることをチェックする

- 備考

このルールはクライアントサイドでの入力チェックはサポートしていない。

◆ dateRange

入力文字列が指定したフォーマットで日付型に変換でき、指定した日付範囲内であることをチェックする。

- 入力チェック設定ファイル (validation.xml) に設定を要する<var>要素

var-name	var-value	必須性	概要
startDate	開始日付	false	日付範囲の開始の閾値となる日付。datePattern または datePatternStrict で指定した日付フォーマットと一致すること。
endDate	終了日付	false	日付範囲の終了の閾値となる日付。datePattern または datePatternStrict で指定した日付フォーマットと一致すること。
datePattern	日付パターン	false	日付のパターンを示す文字列。Date 型のフォーマットルールに従う。
datePatternStrict	日付パターン	false	日付パターンのチェックを厳密に行うかどうか。日付パターンが yyyy/MM/dd の場合、2001/1/1 はエラーとなる。datePattern が指定されている場合、datePattern で指定されたフォーマットが優先される。

- 入力チェック設定ファイル (validation.xml) の例

```

.....
<form name="/validate">
  <field property="userName"
    depends="dateRange">
    <arg key="label.userName" position="0" />
    <arg key="{var:startDate}" position="1" resource="false" />
    <arg key="{var:endDate}" position="2" resource="false" />
    <var>
      <var-name>startDate</var-name>
      <var-value>2000/1/1</var-value>
    </var>
    <var>
      <var-name>endDate</var-name>
      <var-value>2010/12/31</var-value>
    </var>
    <var>
      <var-name>datePattern</var-name>
      <var-value>yyyy/MM/dd</var-value>
    </var>
  </field>
</form>
.....

```

デフォルトのメッセージは「errors.dateRange={0}には{1}から{2}までの範囲で入力してください。」

入力された日付が 2000/1/1~2010/12/31 の範囲内であることをチェックする。

◆ multiFieldCheck

入力された文字列がアクションフォームの他のプロパティの値に依存する等の、複数フィールド関連チェックを行う。関連チェックを行なうクラスは、TERASOLUNA Server Framework for Java (Web 版)が提供する MultiFieldValidator インタフェースを実装して、業務開発者が作成する。

- 入力チェック設定ファイル (validation.xml) に設定を要する<var>要素

var-name	var-value	必須性	概要
fields	検証に必要な他のプロパティ名	false	複数のフィールドを指定する場合はフィールド名をカンマ区切りで 指定する。
multiFieldValidator	MultiFieldValidator 実装クラス名	false	複数フィールドの関連チェックを行う MultiFieldValidator 実装クラス名。

- 入力チェック設定ファイル (validation.xml) の例

```
<form name="/validate">
  <field property="value" depends="multiField">
    <msg key="errors.multiField"
        name="multiField"/>
    <arg key="label.value" position="0" />
    <arg key="label.value1" position="1" />
    <arg key="label.value2" position="2" />
    <var>
      <var-name>fields</var-name>
      <var-value>value1,value2</var-value>
    </var>
    <var>
      <var-name>multiFieldValidator</var-name>
      <var-value>sample.SampleMultiFieldValidator</var-value>
    </var>
  </field>
</form>
```

このルールにはデフォルトのエラーメッセージが存在しないため、必ずメッセージ設定を行う。

検証に必要なその他のプロパティ名

関連チェックを実装したクラス名

- MultiFieldValidator 実装クラスの例

```
public class SampleMultiFieldValidator implements MultiFieldValidator {
  public boolean validate(String value, String[] fields) {

    int value0 = Integer.parseInt(value);
    int value1 = Integer.parseInt(fields[0]);
    int value2 = Integer.parseInt(fields[1]);
    return (value1 <= value0 && value2 >= value0);
  }
}
```

第一引数には検証対象の値、第二引数には検証に必要な他のフィールドの値が渡される。

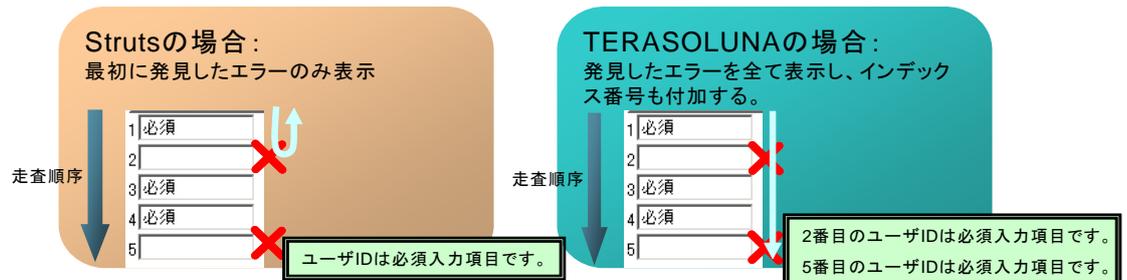
検証結果は boolean 型。エラーの場合は false を返却すること。

- 備考

このルールはクライアントサイドでの入力チェックはサポートしていない。

◆ 配列/Collection プロパティの入力チェック

Struts、commons-validator では繰り返し項目の入力チェックは可能であるが、入力チェックエラーが発生した時点で検証処理が終了してしまうため、TERASOLUNA Server Framework for Java (Web 版)で拡張して、繰り返し項目の入力チェックの途中でエラーが発生しても、全ての要素をチェックできるように拡張している。



配列/Collection プロパティのチェックに対応するルールは以下の通りである。

ルール名				
requiredArray	requiredIfArray	validWhenArray	minLengthArray	maxLengthArray
maskArray	byteArray	shortArray	integerArray	longArray
floatArray	doubleArray	dateArray	intRangeArray	doubleRangeArray
floatRangeArray	creditCardArray	emailArray	urlArray	alphaNumericStringArray
hankakuKanaStringArray	hankakuStringArray	zenkakuStringArray	zenkakuKanaStringArray	capAlphaNumericStringArray
numberArray	numericStringArray	prohibitedArray	stringLengthArray	dateRangeArray
byteLengthArray	byteRangeArray			

※ 各ルールの検証内容・設定方法はそれぞれのルール名から「Array」を除いたルールと同様であるため、各ルール、または Struts のリファレンスを参照のこと。

- 入力チェック設定ファイル (validation.xml) の例

```
.....
<form name="/validateArray">
  <field property="values" depends="stringLengthArray">

    <arg key="##INDEX" position="0" resource="false"/>

    <arg key="label.values" position="1" />
    <arg key="{var:stringLength}" position="2" resource="false" />
    <var>
      <var-name>stringLength</var-name>
      <var-value>3</var-value>
    </var>
  </field>
</form>
.....
```

##INDEX キーワードを使用すると、インデックスの順番をメッセージに使用できる

デフォルトのメッセージは本来のルールで使用するメッセージキー+Array
この場合は「errors.stringLengthArray={0}行目の{1}は{2}文字で入力してください。」

検証ルールの設定方法は本来のルールと同様

※property 属性に指定するプロパティ名は JXPathIndexedBeanWrapperImpl の仕様に従いネストしたプロパティ名を記述することが可能である。

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC20 入力チェック拡張」
 - ◇ /webapps/validation/*
 - ◇ /webapps/WEB-INF/validation/*
 - ◇ jp.terasoluna.thin.functionsample.validation.*
- TERASOLUNA Server Framework for Java (Web 版) チュートリアル
 - 「2.8.1 単項目チェック」
 - 「2.8.2 関連チェック」
 - /webapps/WEB-INF/validation.xml

■ 備考

- なし

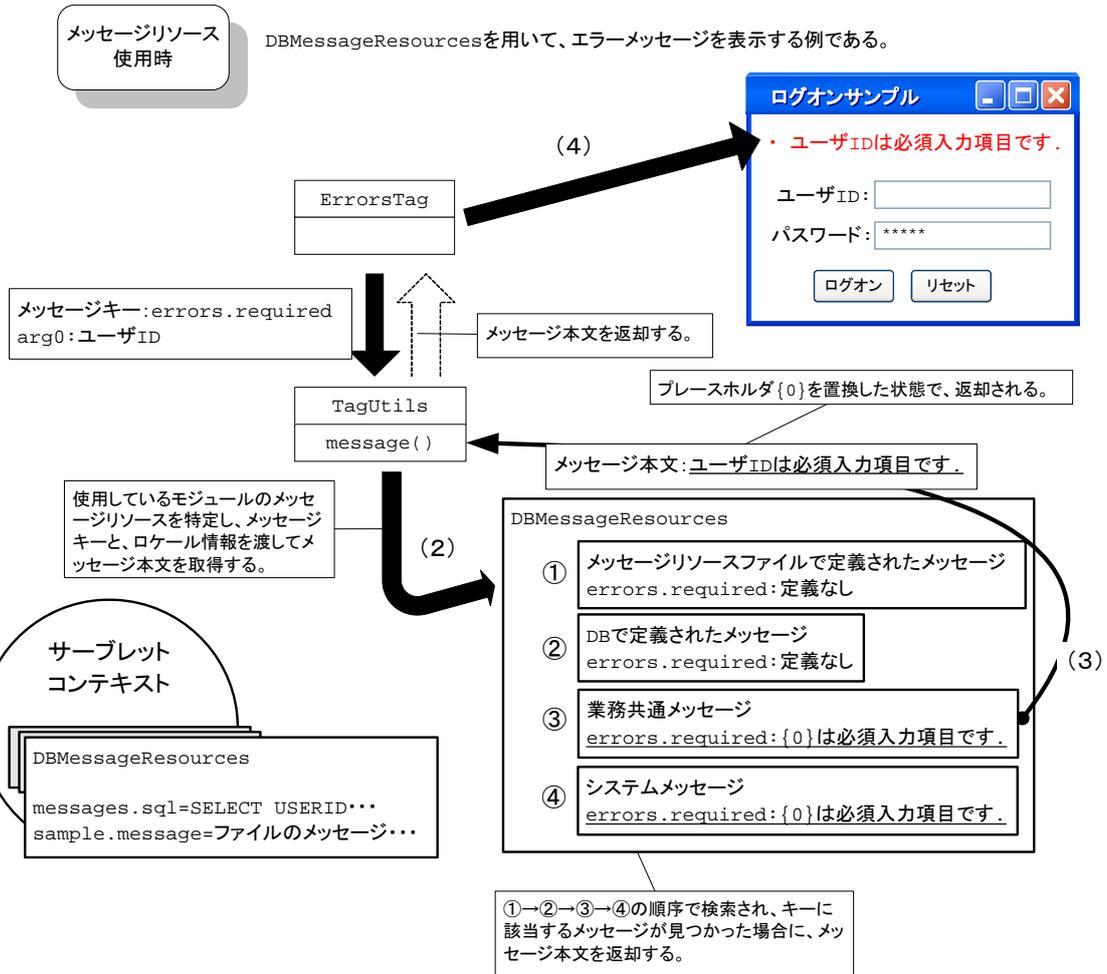
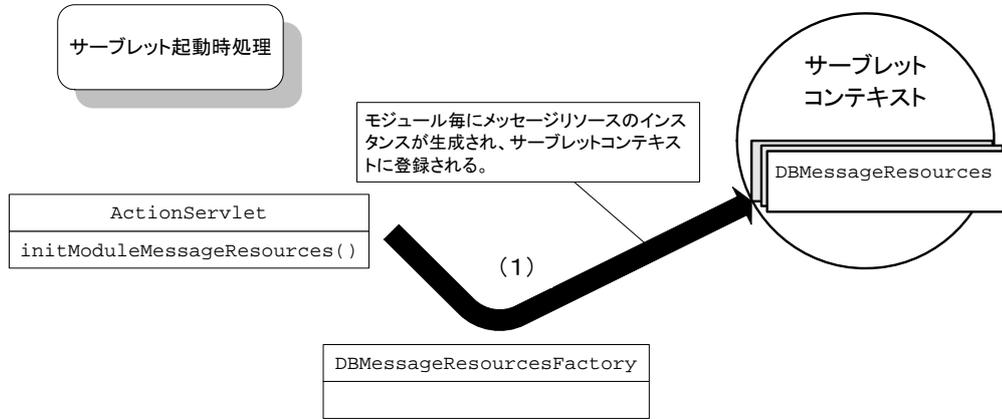
WG-01 拡張メッセージリソース機能

■ 概要

◆ 機能概要

- モジュールごとに別途定義が必要となる Struts のメッセージリソースを拡張し、DB やモジュールをまたがる業務共通メッセージ、システムエラー等の TERASOLUNA Server Framework for Java (Web 版)が規定するデフォルトメッセージを提供する。
- Struts とは異なり、メッセージの定義に優先順位をつけて定義できる。

◆ 概念図



◆ 解説

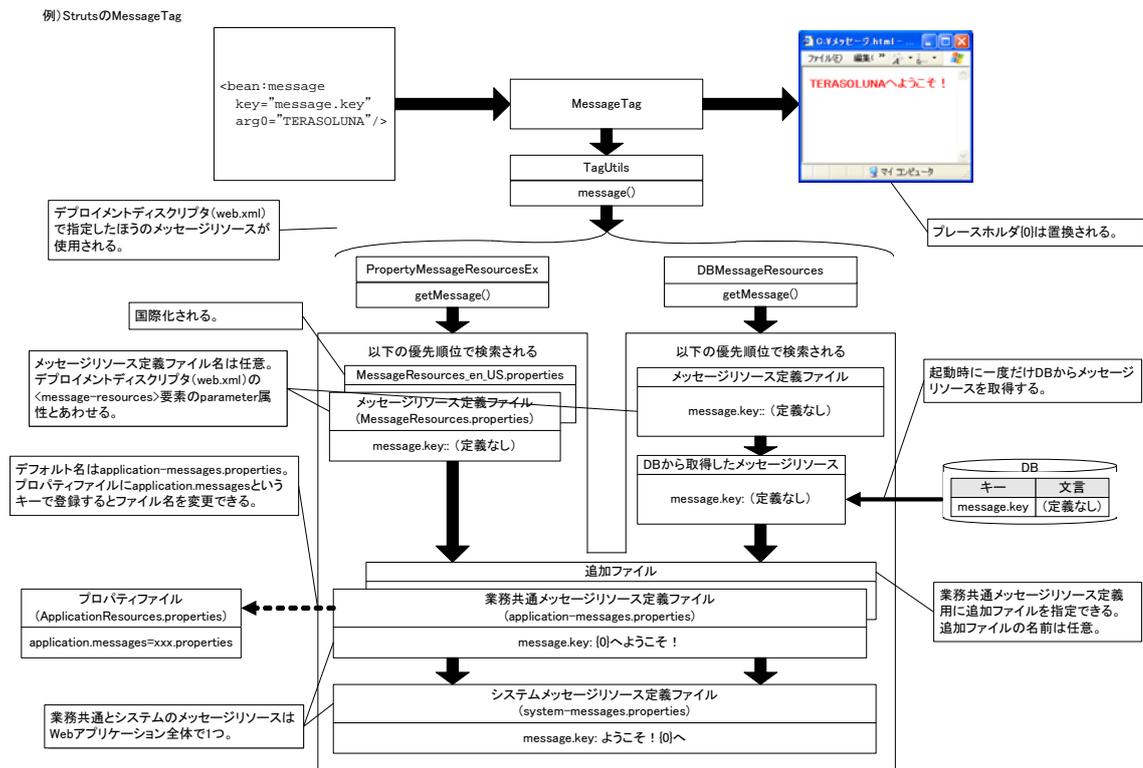
- (1) サブレット起動時に、メッセージリソースファクトリクラスを用いてメッセージリソースインスタンスを生成し、サブレットコンテキストに格納する。(1 モジュールで複数個のメッセージリソースを格納することも可能である。)
- (2) ビジネスロジックでエラーが起こった際の画面表示処理などで、メッセージリソースで定義されたメッセージが必要な場合は、TagUtil 等を通じて、メッセージリソースからメッセージを取得することができる。
- (3) メッセージリソースは定義場所に応じて優先順位が付けられており、優先順位の高い場所に定義されたものが使用される。
- (4) 取得したメッセージを画面に表示する。

■ 使用方法

◆ コーディングポイント

拡張メッセージリソース機能の提供するメッセージリソースは以下の 2 系統がある。

- 拡張プロパティメッセージリソース
プロパティファイルベースで定義する場合
 - DB メッセージリソース
DB でメッセージを定義する場合
- メッセージリソース定義の優先順位
メッセージリソースは以下の場所に定義できる。優先順位順に、
 1. Struts 標準のメッセージリソース定義ファイル (MessageResources.properteis)
 2. DB メッセージリソース定義
 3. 業務共通メッセージリソース定義ファイル (application-messages.properties)
 4. システムメッセージリソース定義ファイル (system-messages.properties)である。



- Struts 標準のメッセージリソース定義ファイル (MessageResources.properties)
Struts 標準のメッセージリソース定義はプロパティファイル形式のメッセージリソースを利用するものである。

メッセージリソース定義ファイルのファイル名は任意だが **ApplicationResources.proeprieties** にはしないこと。これは『CD-01 ユーティリティ機能』のうち、PropertyUtil で使われるプロパティファイル名だからである。一般的に、メッセージリソース定義ファイルのファイル名には MessageResources.properties を用いる。

なお、1.の拡張プロパティメッセージリソースを用いる場合、Struts 標準のメッセージリソース定義ファイルは国際化可能である。

ただし、このメッセージリソースに無かったキーは業務共通メッセージリソース定義ファイル (application-messages.properties)、およびシステムメッセージリソース定義ファイル (system-messages.properties) から検索されるが、それらは国際化されないので注意すること。

- DB メッセージリソース定義
DB メッセージリソースを用いる場合、DB からメッセージリソースを取得できる。

DB へのアクセスには MessageResourcesDAO インタフェース実装クラスを利用する。一実装として、Spring フレームワークを利用した MessageResourcesDAO 実装

(MessageResourcesDaoImpl) を提供しているので、以下ではそれを利用した設定について説明する。

MessageResourcesDaoImpl を利用するには、

1. データソースの設定
2. データを取得する SQL の定義
3. MessageResourcesDaoImpl を利用するという宣言

の3点を設定する必要がある。

このうち、1は dbMessageResources.xml に、2と3はシステム設定プロパティファイル (system.properties) に書く。

➤ Bean 定義ファイル (dbMessageResources.xml)

```
<bean id="dataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>jdbc/sample</value>
  </property>
</bean>
```

JndiObjectFactoryBean を指定する。

JNDI 名を指定する。

dbMessageResources.xml はクラスパスの通った場所に配備すること。

なお、dbMessageResources.xml は Bean 定義ファイルだが、ビジネスロジックの管理に用いる Bean 定義ファイルとは別であり、データソースの定義のみを記述する。

ファイル名 dbMessageResources.xml は固定である。

➤ システム設定プロパティファイル (system.properties)

```
messages.sql=SELECT MESSAGE_KEY, MESSAGE_VALUE FROM MESSAGES
messages.dao=jp.terasoluna.fw.web.struts.action.MessageResourcesDaoImpl
```

メッセージリソースを取得する SQL を指定する。

完全修飾名で MessageResourcesDAO の実装クラスを指定する。

MessageResourcesDaoImpl では、DB にメッセージリソースを定義する場合、カラムを2つ取得する SQL 文を定義する。このとき、第一カラムの値がメッセージキー、第二カラムの値がメッセージ文言として扱われる。DB のカラム名やテーブル名に特に制限はない。

- 業務共通メッセージリソース定義ファイル (application-messages.properties)
業務共通メッセージリソース定義ファイルでは Struts のモジュール分割機能によって分割されたモジュールすべてが共有するメッセージリソースを定義する。

ファイル名はデフォルトで application-messages.properties が用いられるが、システム設定プロパティファイル (system.properties) に application.messages をキーとして任意のファイル名を指定すると、そのファイルが使用される。

➤ システム設定プロパティファイル (system.properties)

```
application.messages=another-application-messages
```

必ず、**properties** は除くこと。

また、業務共通メッセージリソース定義ファイルを追加用の別ファイルに分けることができる。その場合、以下のように設定する。

➤ 業務共通メッセージリソース定義ファイル (application-messages.properties)

```
add.message.file.1=additional-messages
```

```
add.message.file.2=another-additional-messages
```

追加用ファイルの名前・数は任意。

追加するファイル名のプロパティキーは
add.message.file.<通番>

```
message=メッセージ
```

別ファイルに分けることとは関係なく、直接ここにメッセージリソースを定義できる。

➤ 追加用メッセージリソース定義ファイル (additional-messages.properties)

```
additional.message=追加用メッセージ
```

追加用メッセージリソース定義ファイルから、さらにファイルの追加はできない。

※ <通番>となっている部分は必ず「1」から順に連続した番号を振ること。
途中の番号が欠けている場合、欠けた番号以降は無効となる。

● システムメッセージリソース定義ファイル (system-messages.properties)

システムメッセージリソース定義ファイル (system-messages.properties) では Struts のモジュール分割機能によって分割されたモジュールすべてが共有するメッセージリソースが定義されている。システムメッセージリソースは、TERASOLUNA Server Framework for Java (Web 版)が出力するシステムエラーメッセージについて、あらかじめデフォルト値として定義されている。

原則として業務開発者は変更しなくて良い。

業務要件により、システムエラーメッセージの値を変更したい場合は、システムメッセージリソース定義ファイル (system-messages.properties) を変更するのではなく、業務共通メッセージリソース定義ファイル (application-messages.properties) 等に同じキーで登録する。

なお、業務共通メッセージリソース定義ファイル (application-messages.properties) とは異なり、system-messages.properties というファイル名は固定である。

- 拡張メッセージリソースの提供するメッセージリソースの設定方法
拡張メッセージリソース機能では、2 系統のメッセージリソースを提供している。以下ではそれぞれの場合の設定方法について説明する。

1. 拡張プロパティメッセージリソースの場合

➤ Struts 設定ファイル(struts-config.xml)

```
<struts-config>
.....
<message-resources
factory="jp.terasoluna.fw.web.struts.action.PropertyMessageResourcesExFactory"
parameter="MessageResources" />
.....
</struts-config>
```

PropertyMessageResourcesExFactory
を指定する。

Struts 標準のメッセージリソース定義ファイル
を指定する。名前は任意でよい。
.properties はつけなくてよい。

2. DB メッセージリソースの場合

➤ Struts 設定ファイル(struts-config.xml)

```
<struts-config>
.....
<message-resources
factory="jp.terasoluna.fw.web.struts.action.DBMessageResourcesFactory"
parameter="MessageResources" />
.....
</struts-config>
```

DBMessageResourcesFactory
を指定する。

Struts 標準のメッセージリソース定義ファイル
を指定する。名前は任意でよい。
.properties はつけなくてよい。

➤ システム設定プロパティファイル (system.properties)

```
messages.sql=SELECT MESSAGE_KEY, MESSAGE_VALUE FROM MESSAGES
messages.dao=jp.terasoluna.fw.web.struts.action.MessageResourcesDaoImpl
```

messages.sql というキーで DB からメッセージキーとメッセージ
文言が格納された結果セットを取得する SQL を定義する。

messages.dao というキーで DB からメッセージリソースを取得する実装
クラスを指定する。
※一実装として MessageResourcesDaoImpl を提供している。

◆ 拡張ポイント

- メッセージリソースを DB から取得する DB メッセージリソースでは、MessageResourcesDAO という DB アクセスを隠蔽するためのインタフェースを用意している。Spring に非依存の形で DB アクセスしたい場合などは、このインタフェースを実装し、TERASOLUNA Server Framework for Java (Web 版)が一実装として用意している MessageResourcesDaoImpl に置き換えて使用できる。ただし、その際データソースにどのようにアクセスするか「データソースの設定」の方法については実装クラスで決める必要がある。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.struts.action.PropertyMessageResourcesEx	Struts の PropertyMessageResources を継承したメッセージリソースクラス。
2	jp.terasoluna.fw.web.struts.action.PropertyMessageResourcesExFactory	PropertyMessageResourcesEx のファクトリクラス。Struts 設定ファイル(struts-config.xml)のメッセージリソースの設定部分にはメッセージリソースクラスではなくファクトリクラスを指定する。
3	jp.terasoluna.fw.web.struts.action.DBMessageResources	DB に設定されたモジュール共通のメッセージを取得できるメッセージリソースクラス。
4	jp.terasoluna.fw.web.struts.action.DBMessageResourcesFactory	DBMessageResources のファクトリクラス。Struts 設定ファイル(struts-config.xml)のメッセージリソースの設定部分にはメッセージリソースクラスではなくファクトリクラスを指定する。
5	jp.terasoluna.fw.web.struts.action.MessageResourcesDAO	DBMessageResources が DB アクセスするためのインタフェース。
6	jp.terasoluna.fw.web.struts.action.MessageResourcesDaoImpl	MessageResourcesDAO を実装した、DBMessageResources が DB アクセスするための一実装クラス
7	jp.terasoluna.fw.web.struts.action.GlobalMessageResources	業務共通・システムメッセージリソースを保持するクラス。

■ 関連機能

- 『CD-01 ユーティリティ機能』

■ 使用例

- Terasoluna Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC21 拡張メッセージリソース」
 - ◇ /webapps/messageex/*
 - ◇ /webapps/WEB-INF/messageex/*
 - ◇ jp.terasoluna.thin.functionsample.messageex.*
 - ◇ /webapps/messageex2/*
 - ◇ /webapps/WEB-INF/messageex2/*
 - ◇ jp.terasoluna.thin.functionsample.messageex2.*
- Terasoluna Server Framework for Java (Web 版) チュートリアル
 - webapps/WEB-INF/struts-config.xml

■ 備考

- なし

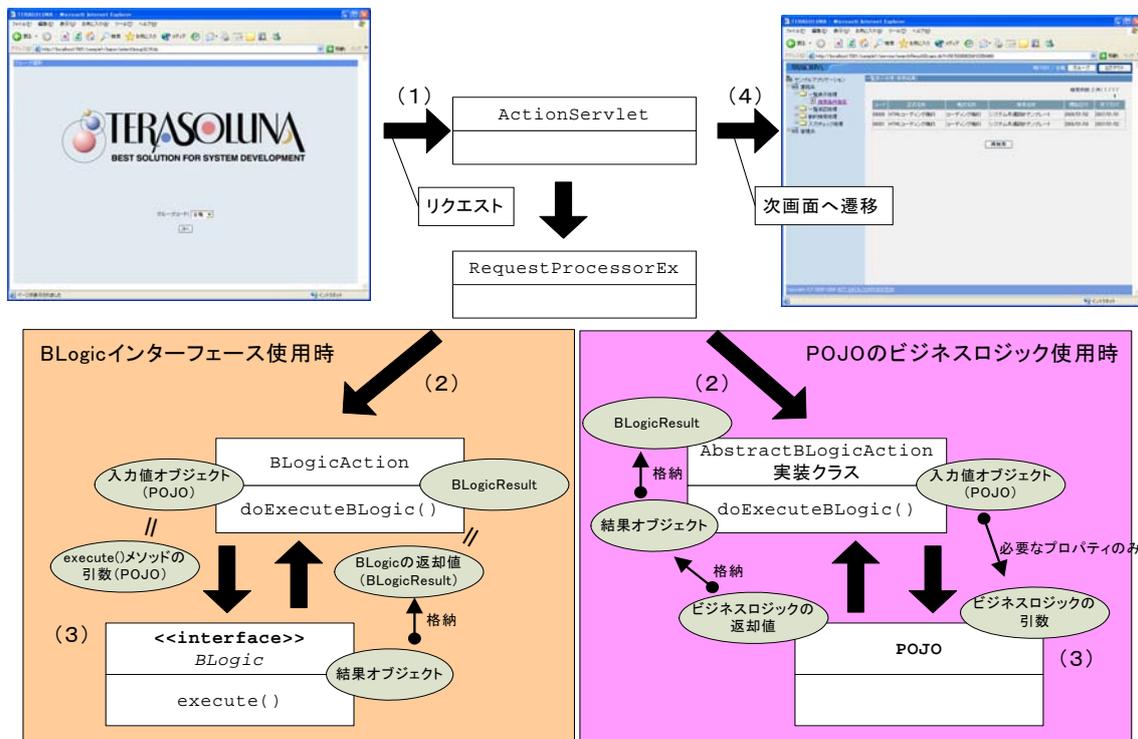
WH-01 ビジネスロジック実行機能

■ 概要

◆ 機能概要

- ビジネスロジックの実行方式を提供する。ビジネスロジッククラスは、BLogic インターフェース実装クラスまたは POJO のいずれかの方式を選択できる。各方式を選択した場合に、業務開発者が実装するクラスは以下の通りである。
 - BLogic インターフェース実装クラスを使用
 - ◇ BLogic インターフェース実装クラス (BLogicAction クラスより起動)
 - POJO を使用
 - ◇ AbstractBLogicAction 実装クラス
 - ◇ POJO のビジネスロジッククラス (AbstractBLogicAction 実装クラスより起動)
- ビジネスロジック実行時に返却される BLogicResult に設定されたメッセージ情報 (BLogicMessages) を Struts 用の ActionMessages インスタンスに変換して、Web 層に設定する。

◆ 概念図



◆ 解説

- (1) ブラウザからリクエストを WebAP サーバに送信する。
- (2) アクションパスに関連付けられたアクションクラスが呼び出される。ビジネスロジック実行機能を使用する場合、アクションクラスには Terasoluna Server Framework for Java (Web 版)が提供する BLogicAction、または業務開発者により実装された AbstractBLogicAction 実装クラスを指定する。
- (3) ビジネスロジックを実行する。

- BLogic インターフェース使用時には、BLogicAction の Bean 定義にて businessLogic プロパティに設定された（業務開発者によって指定された）BLogic 実装オブジェクトをロードし、execute()メソッドを実行する。
- POJO のビジネスロジック使用時には、AbstractBLogicAction 実装クラスの doExecuteBLogic()メソッドから POJO に実装されたビジネスロジックを直接実行する。

返却値である BLogicResult には、ビジネスロジック実行結果を表す文字列、結果オブジェクト、メッセージが格納される。

- (4) BLogicResult に設定されたオブジェクト、メッセージを Web 層に反映した後、ビジネスロジック実行結果を表す文字列から、次に遷移する画面が決定される。

■ 使用方法

◆ コーディングポイント

- BLogic インターフェースを使用

BLogic インターフェースを実装し、ビジネスロジックを作成する。BLogic インターフェースの入力値は POJO で実装し、画面仕様、ビジネスロジックの仕様に従い必要なプロパティを定義する。入力値が存在しないビジネスロジックは入力値が null となる。また、出力値は BLogicResult を使用する。

- BLogic インタフェースの実装例

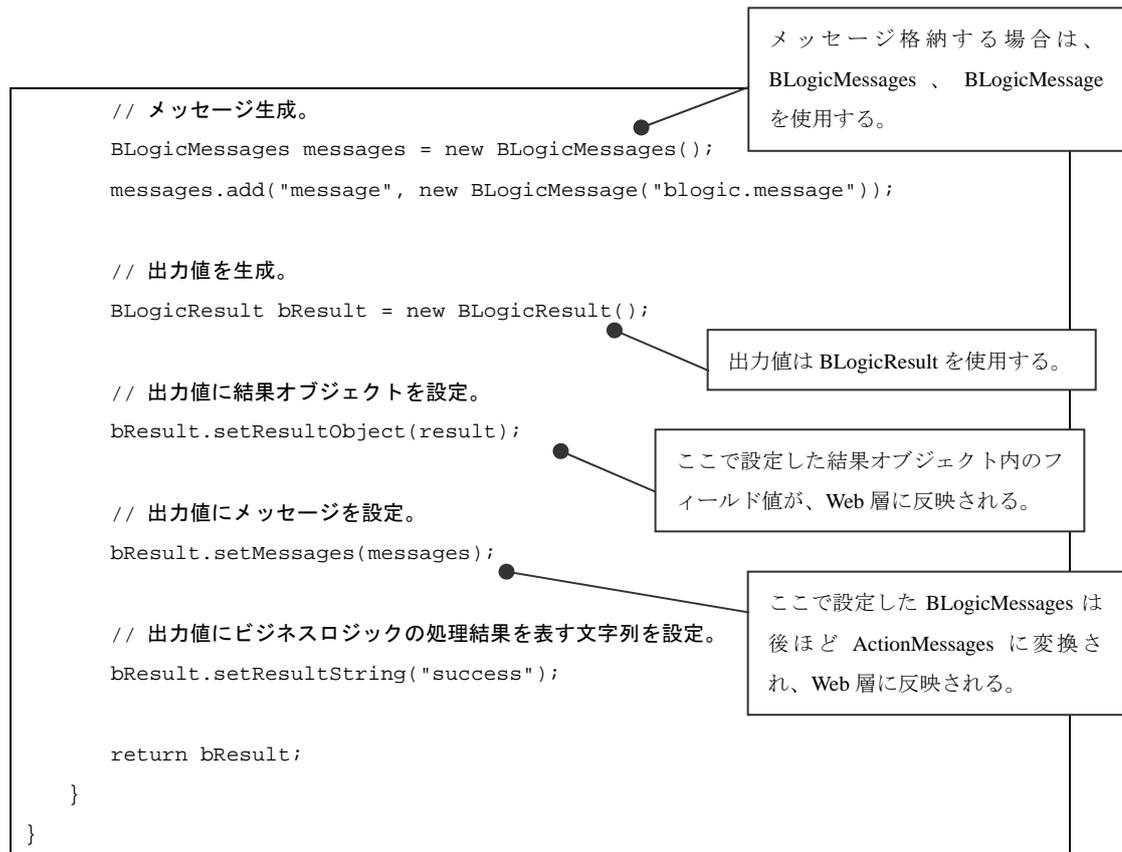
```
public class AddBLogic implements BLogic<AddBLogicParam> {
    public BLogicResult execute(AddBLogicParam params) {
        // 入力値クラスから値を取得する。
        int value1 = params.getValue1();
        int value2 = params.getValue2();

        // 結果オブジェクト。
        AddBLogicResult result = new AddBLogicResult();
        result.setResult(value1 + value2);
    }
}
```

入力値クラスは Generic で型を指定する。入力値がない場合は、Generics 指定しない。

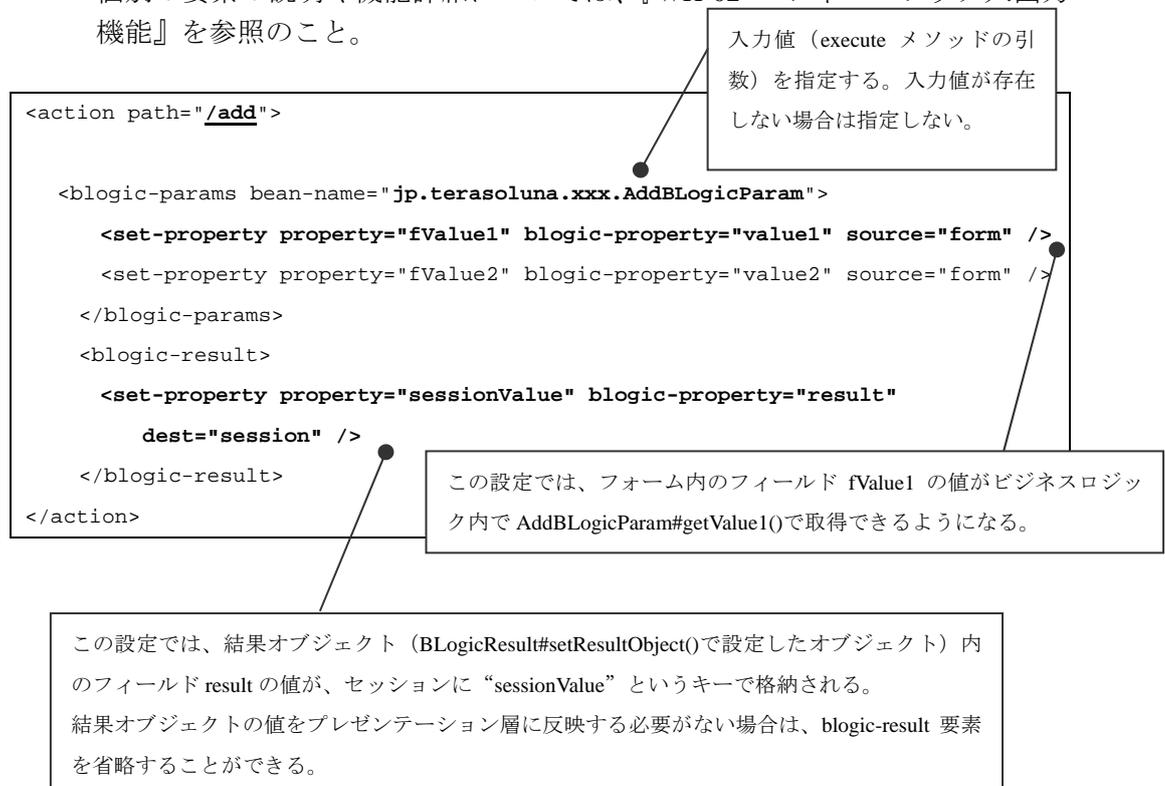
ここでは POJO で実装している。Map 型も使用可。

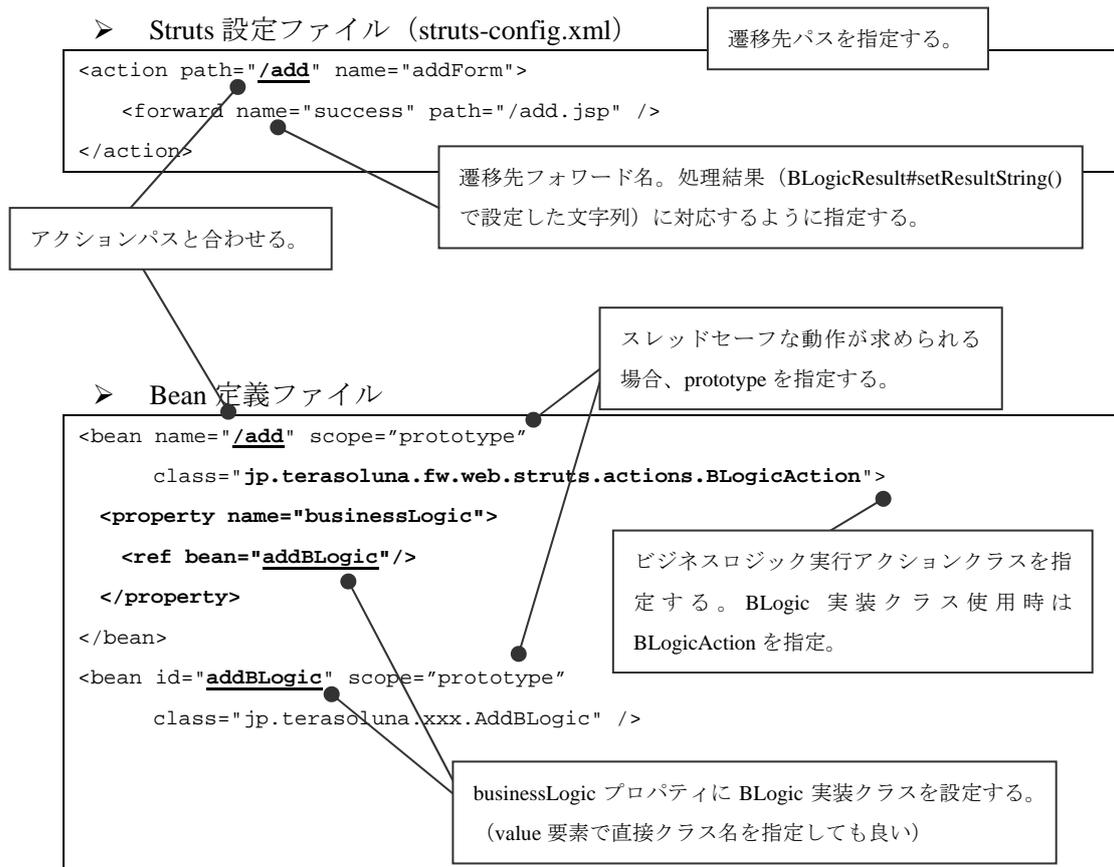
(続く)



➤ ビジネスロジック入出力情報定義ファイル (blogic-io.xml)

個別の要素の説明や機能詳細については、『WH-02 ビジネスロジック入出力機能』を参照のこと。





※モジュール分割を行なう場合は、name 属性には「/モジュール名+アクションパス」を設定すること。

- POJO のビジネスロジッククラスを使用

ビジネスロジックのポータビリティを保つ場合は POJO で実装する必要がある。POJO は、可能な限り TERASOLUNA Server Framework for Java (Web 版) の API に依存しないように実装し、ビジネスメソッドのインターフェースには必要な値のみ指定する。Web 層 (Struts、TERASOLUNA Server Framework for Java (Web 版)とのブリッジのため AbstractBLogicAction を実装したアクションクラスを作成する。

➤ AbstractBLogicAction 実装クラスの実装例

```
public class AddAction
    extends AbstractBLogicAction<AddBLogicParam> {
    // POJOで実装したビジネスロジック。
    private AddBLogic addBLogic = null;

    public BLogicResult doExecuteBLogic(AddBLogicParam param) throws Exception {
        // 入力値クラスから入力値を取得する。
        int value1 = param.getValue1();
        int value2 = param.getValue2();

        // ビジネスロジックの呼び出し。
        int result = addBLogic.add(value1, value2);

        // 結果オブジェクト。
        Map<String, Integer> map = new HashMap<String, Integer>();
        map.put("result", result);

        // 出力値を生成。
        BLogicResult bResult = new BLogicResult();

        // 出力値に結果オブジェクトを設定。
        bResult.setResultObject(map);

        // 出力値にビジネスロジックの処理結果を表す文字列を設定。
        bResult.setResultString("success");

        return bResult;
    }
}
```

doExecuteBLogic の引数の型は Generics で指定する。入力値が存在しない場合は Generics 指定しない。

DI コンテナで設定する。
※ここでは getter/setter は省略

Action クラスの中ではビジネスロジックは記述しないこと。

ここでは Map 型で実装している。POJO も使用可。

doExecuteBLogic の戻り値は BLogic インターフェースの戻り値と同様に BLogicResult を使用。

➤ POJO の実装例

インターフェースベースの実装を行なうために、POJO に対してインターフェースを作成する。

```
public interface AddBLogic {
    int add(int value1, int value2);
}
```

作成したインターフェースを実装した POJO を作成する。

```
public class AddBLogicImpl implements AddBLogic
{
    public int add(int value1, int value2) {
        return value1 + value2;
    }
}
```

Web 層、プレゼンテーション層、Struts、TERASOLUNA に依存しないように実装することで、このビジネスロジックがどんなアプリケーションでも使用可能なクラスとなる。

ビジネスロジックは TERASOLUNA Server Framework for Java (Web 版)の API に依存しないように実装するべきだが、データベースアクセスを行なう場合は、ビジネスロジック内で TERASOLUNA Server Framework for Java (Web 版)の DAO クラスを使用する必要がある。データベースアクセスの詳細については、『CB-01 データベースアクセス機能』を参照のこと。

➤ ビジネスロジック入出力情報定義ファイル (blogic-io.xml)
BLogic インターフェース使用時と同様。

➤ Struts 設定ファイル (struts-config.xml)

```
<action path="/add" name="addForm">
    <forward name="success" path="/add.jsp" />
</action>
```

➤ Bean 定義ファイル

```
<bean name="/add" scope="prototype"
    class="jp.terasoluna.xxx.AddAction">
    <property name="addBLogic">
        <ref bean="addBLogic"/>
    </property>
</bean>
<bean id="addBLogic" scope="prototype"
    class="jp.terasoluna.xxx.AddBLogicImpl" />
```

AbstractBLogicAction 実装クラスを設定する。

AddAction の AddBLogic プロパティに POJO のビジネスロジックを設定する。(value 要素で直接クラス名を指定しても良い)

その他の設定は BLogic インターフェース使用時と同様。

● トランザクション管理

トランザクションの制御方法については、『CA-01 トランザクション管理機能』を参照のこと。

◆ 拡張ポイント

なし。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.struts.actions.AbstractBLogicAction	ビジネスロジックの起動を行うアクションクラスに共通する機能を集約した抽象クラス。
2	jp.terasoluna.fw.web.struts.actions.BLogicAction	BLogic 起動クラス。AbstractBLogicAction を実装している。
3	jp.terasoluna.fw.service.thin.BLogic	BLogicAction から起動するビジネスロジックが実装すべきインターフェース。
4	jp.terasoluna.fw.service.thin.BLogicResult	ビジネスロジックからの出力情報を保持するクラス。
5	jp.terasoluna.fw.service.thin.BLogicMessage	メッセージ情報クラス。
6	jp.terasoluna.fw.service.thin.BLogicMessages	メッセージ情報一覧クラス。BLogicMessage インスタンスを格納する。

■ 関連機能

- 『CA-01 トランザクション管理』
- 『CB-01 データベースアクセス機能』
- 『WH-02 ビジネスロジック入出力機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC22 ビジネスロジック実行」
 - ◇ /webapps/blogic/*
 - ◇ /webapps/WEB-INF/blogic/*
 - ◇ jp.terasoluna.thin.functionsample.blogic.*
- TERASOLUNA Server Framework for Java (Web 版) チュートリアル
 - 「2.4 ログオン」
 - /webapps/WEB-INF/moduleContext.xml

■ 備考

- なし。

WH-02 ビジネスロジック入出力機能

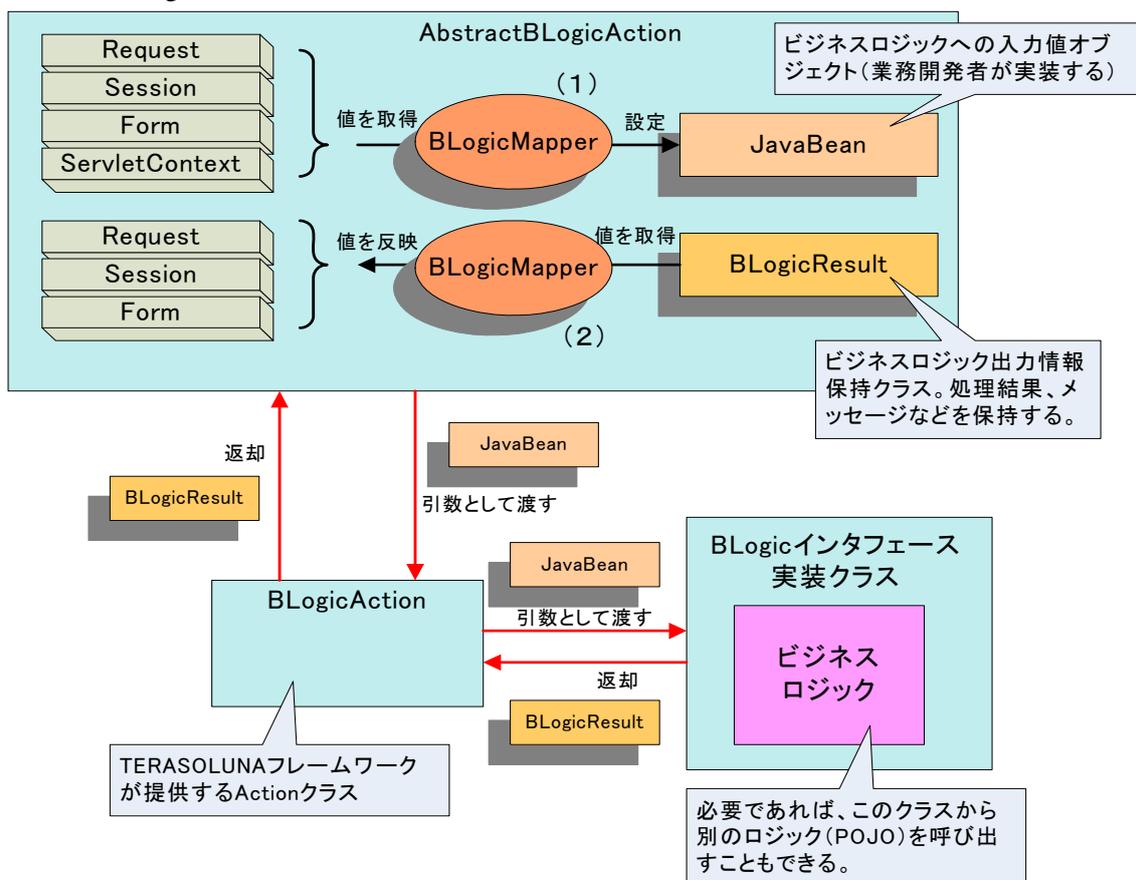
■ 概要

◆ 機能概要

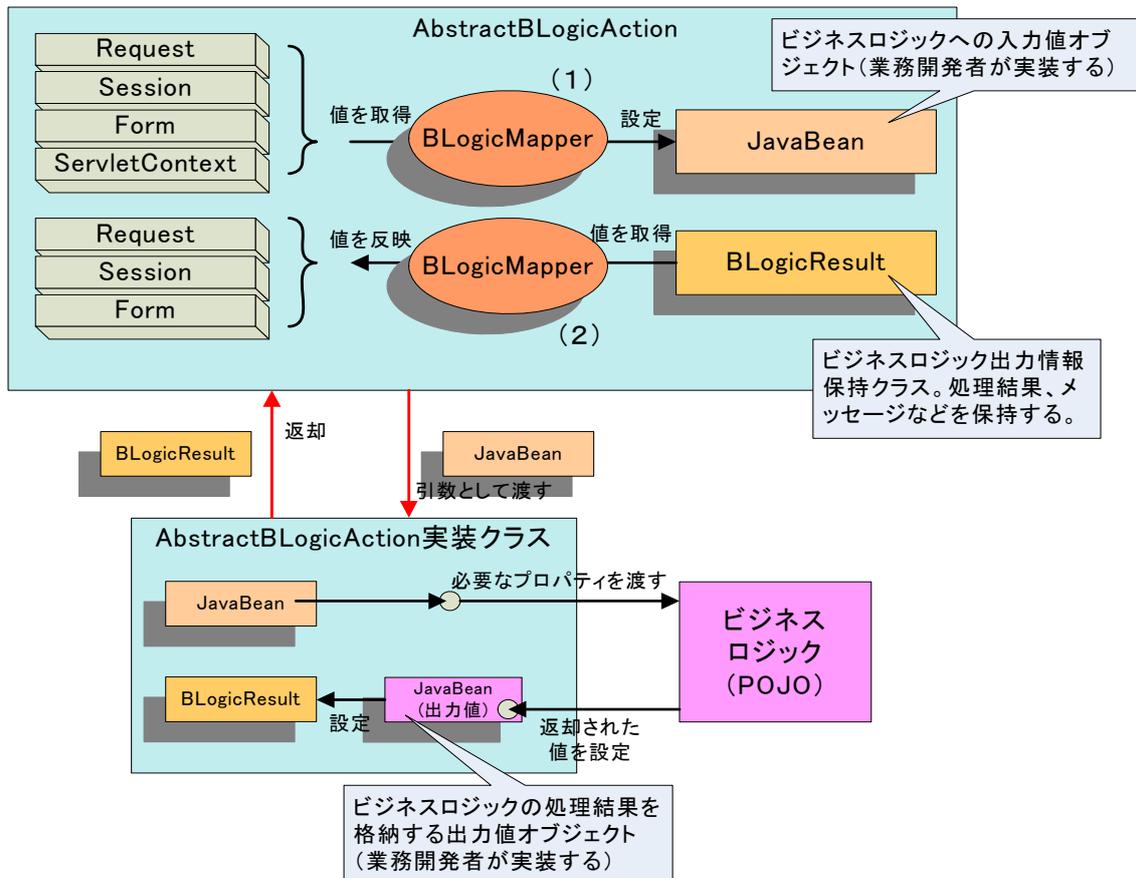
- 設定ファイルに入出力の設定を記述することで、プレゼンテーション層⇄サービス層間のデータの変換を自動化する。
- ビジネスロジック入出力設定の初期化には、Struts のプラグインの機能を利用する。
- デフォルトでは、リクエスト、セッション、アクションフォーム、サーブレットコンテキストからビジネスロジックへの入力処理、ビジネスロジックからリクエスト、セッション、アクションフォームへの出力処理を行える。

◆ 概念図

- BLogicAction 使用時の入出力情報の流れ



● AbstractBLogicAction 実装クラス使用時の入出力情報の流れ



◆ 解説

● 初期化について

- 入出力情報定義の初期化処理は、サーブレット起動時に、プラグインとして実行される。ビジネスロジック入出力情報定義ファイル(`blogic-io.xml`)を読み込み、設定情報をサーブレットコンテキストに登録する。ビジネスロジックへの入出力設定の際には、現在のアクションパスに対応する設定情報をサーブレットコンテキストから取得して使用する。

● 入出力設定処理

- (1) ビジネスロジック実行アクションが呼び出されると、`BLogicMapper` は `blogic-io.xml` の入力情報設定に従い、プレゼンテーション層からビジネスロジックに対する入力値オブジェクトの生成に必要なオブジェクトを収集する。収集したオブジェクトを入力情報設定に指定された名前を入力値オブジェクトに格納し、それを引数にビジネスロジック実行アクション (`BLogicAction` または `AbstractBLogicAction` 実装クラス) を実行する。入力値がないビジネスロジックの場合は `blogic-io.xml` の `bean-name` 属性を省略することで、入力値オブジェクトが `null` の状態でビジネスロジックを実行する。
- (2) ビジネスロジック実行後、実行結果が格納された `BLogicResult` が返却されると、

BLogicMapper は `blogic-io.xml` の出力情報設定に従い、`BLogicResult` から値を取得し、プレゼンテーション層に反映する。プレゼンテーション層への出力値反映が不要な場合は、`blogic-result` 要素を省略することができる。

ビジネスロジックの実行についての詳細は、『WH-01 ビジネスロジック実行機能』を参照のこと。

■ 使用方法

◆ コーディングポイント

ビジネスロジック入出力機能において、実装すべきポイントとして下記の項目が挙げられる。

1. プレゼンテーション層のオブジェクトの設定
2. ビジネスロジック入出力情報定義ファイルの作成
3. ビジネスロジック内部での入力情報の取得処理
4. ビジネスロジック内部での出力情報の反映処理

ここでは、具体的な例としてアクションフォーム (`DynaValidatorActionFormEx`) 内の `userId`、`password` フィールドで指定された値を用いて DB からユーザ情報を取り出し、以下の場合を例に実装を行う。

1. ユーザ情報が DB に存在する場合は、ユーザバリューオブジェクトを作成し、セッションに格納する。また、成功ステータスとして `BLogicResult` の `resultString` (ビジネスロジックの実行結果を表すフィールド) には `success` を設定して返却する。
2. ユーザ情報が DB に存在しない場合は、`BLogicResult` にメッセージを格納する。失敗ステータスとして、`BLogicResult` の `resultString` には `failure` を設定して返却する。

- ビジネスロジック入出力情報定義ファイル (blogic-io.xml)
 - blogic-io.xml 上部には、Digester で XML からオブジェクトにパースする際に検証ルールとして使用する DTD を下記の通りに記述する。
記述ルールは固定である為、拡張を行なわない限り変更の必要はない。

```

<?xml version="1.0" encoding="Windows-31J" ?>
<!DOCTYPE blogic-io [
  <!ELEMENT blogic-io      (action*)>
  <!ELEMENT action        (blogic-params?,blogic-result?)>
  <!ATTLIST action         path          CDATA #REQUIRED>
  <!ELEMENT blogic-params (set-property*)>
  <!ATTLIST blogic-params  bean-name     CDATA #IMPLIED>
  <!ELEMENT blogic-result (set-property*)>
  <!ELEMENT set-property  EMPTY>

  <!ATTLIST set-property  property      CDATA #REQUIRED>
  <!ATTLIST set-property  blogic-property CDATA #IMPLIED>
  <!ATTLIST set-property  source        CDATA #IMPLIED>
  <!ATTLIST set-property  dest         CDATA #IMPLIED>
]

```

ビジネスロジック実行機能を使用するアクションパスごとに action 要素を設定する。

ビジネスロジック入力情報設定。bean-name 属性にはビジネスロジックへの入力値オブジェクトを指定する。入力値がない場合は bean-name は省略する。set-property 要素を入力項目の数だけ記述する。

ビジネスロジック出力情報設定。set-property 要素を出力項目の数だけ記述する。出力値反映がない場合は blogic-result 要素は省略する。

ビジネスロジック入出力値に対して個別に set-property 要素を設定する。

property 属性・・・値の取得元のフィールド名を指定する。

blogic-property 属性・・・ビジネスロジックで用いる入出力情報のキーを指定する。
省略すると、property 属性と同様の値で処理が実行される。

source 属性・・・値の取得元を指定する。(blogic-params 要素で必要。)

取得元がアクションフォームの場合「form」、セッションの場合「session」、リクエストの場合「request」、サーブレットコンテキストの場合「application」を記述する。AbstractBLogicMapper を拡張することでその他の入力を定義することも可能。

dest 属性・・・値の反映先を指定する。(blogic-result 要素で必要。)

反映先がアクションフォームの場合「form」、セッションの場合「session」、リクエストの場合「request」を記述する。AbstractBLogicMapper を拡張することでその他の出力を定義することも可能。

□デフォルトではサーブレットコンテキストへの出力は対象としていない。

➤ 以下に `blogic-io.xml` 本文の記述例を示す。

```
<blogic-io>
  <action path="/logon">
    <blogic-params bean-name="jp.terasoluna.xxx.LogonBLogicParam">
      <set-property property="userId" blogic-property="uid" source="form" />
      <set-property property="password" blogic-property="pwd" source="form" />
    </blogic-params>
    <blogic-result>
      <set-property property="USER_VALUE_OBJECT" blogic-property="uvo" dest="session" />
    </blogic-result>
  </action>
  .....
</blogic-io>
```

アクションパス
に合わせる。

ビジネスロジックへの入力
値オブジェクトを指定す
る。入力値がない場合は
bean-name は省略する。

アクションフォームの `userId` フィールドを、
`uid` として、ビジネスロジックに渡す。指定さ
れたフィールドがフォームに存在しなければな
らない。

ビジネスロジックからの返却値 `uvo`
を、`USER_VALUE_OBJECT` をキー
に、セッションに格納する。出力値反
映がない場合は `blogic-result` 要素は省
略する。

- Struts 設定ファイル (`struts-config.xml`)

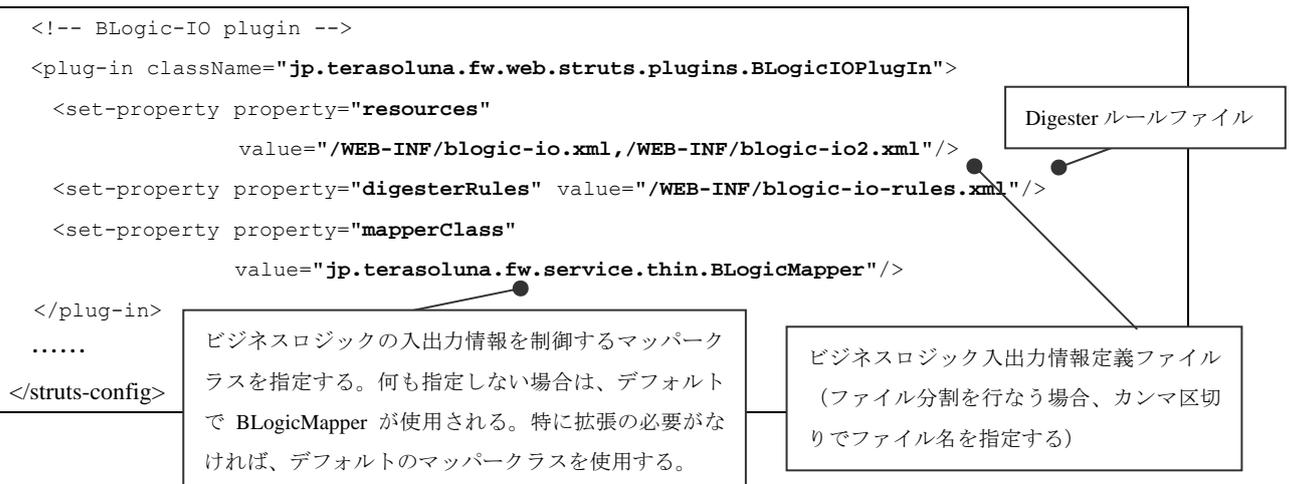
➤ `plug-in` 要素として `BLogicIOPlugin` を指定することで、ビジネスロジック入出力機能を使用できる。

```
<struts-config>
  <form-beans>
    <form-bean name="logonForm"
      type="jp.terasoluna.fw.web.struts.form.DynaValidatorActionFormEx">
      <form-property name="usrId" type="java.lang.String"/>
      <form-property name="password" type="java.lang.String"/>
      .....
    </form-bean>
    .....
  </form-beans>

  <action-mappings type="jp.terasoluna.fw.web.struts.action.ActionMappingEx">
    <!-- ログオン処理 -->
    <action path="/logon" name="logonForm" scope="request">
      <forward name="success" path="/selectGroupSCR.do"/>
      <forward name="failure" path="/logonSCR.do"/>
    </action>
  </action-mappings>
  .....
</struts-config>
```

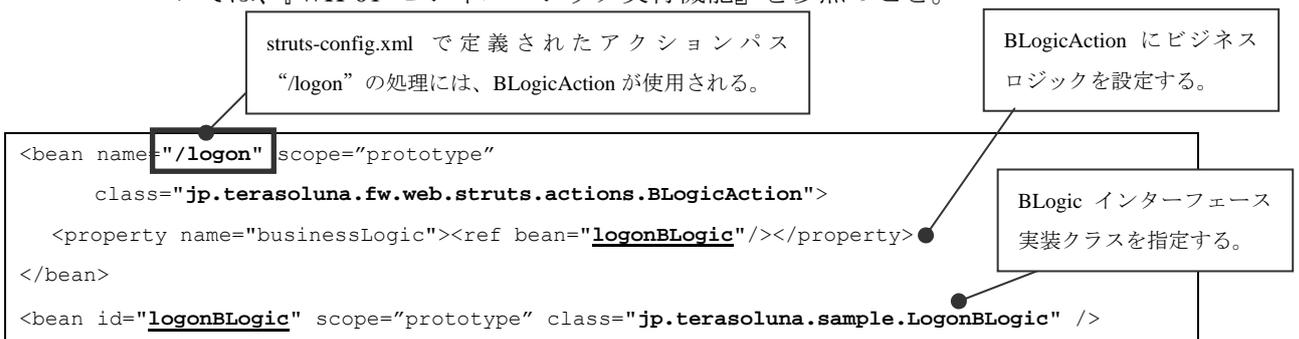
`form-bean` 要素にて定義したアクションフォ
ーム名を指定する。

(続く)



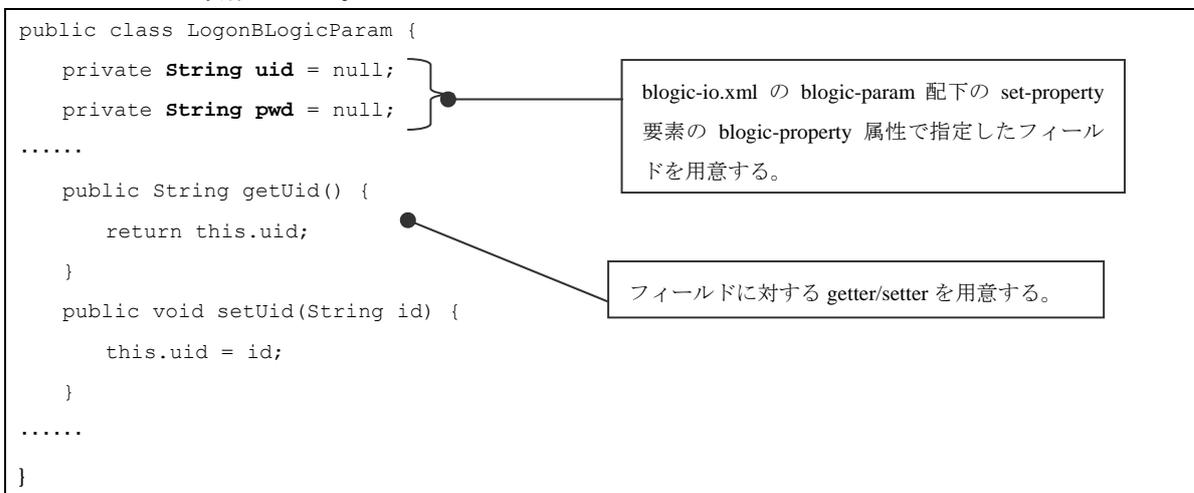
- Bean 定義ファイル

- ここではビジネスロジック実行アクションクラスに BLogicAction を用いる。BLogicAction に対して、BLogic インターフェース実装クラス LogonBLogic を設定する。ビジネスロジック実行における Bean 定義ファイル記述方法については、『WH-01 ビジネスロジック実行機能』を参照のこと。



- ビジネスロジック入力値オブジェクト (LogonBLogicParam.java)

- プレゼンテーション層から、ビジネスロジックに渡す値を格納するためのオブジェクトであり、業務開発者が実装する。LogonBLogic#execute()メソッドの引数となる。



- ビジネスロジック出力値オブジェクト (LogonBLogicResult.java)
 - ビジネスロジックの処理結果の値を格納するためのオブジェクトであり、業務開発者が実装する。LogonBLogic#execute()メソッドの戻り値 BLogicResult の resultObject に格納して返却する。

```
public class LogonBLogicResult {
    private SampleUserValueObject uvo = null;
    .....
    public SampleUserValueObject getUvo() {
        return this.uvo;
    }
    .....
    public void setUvo(SampleUserValueObject obj) {
        this.uvo = obj;
    }
    .....
}
```

blogic-io.xml の `blogic-result` 配下の `set-property` 要素の `blogic-property` 属性で指定したフィールドを用意する。

フィールドに対する `getter/setter` を用意する。

- ビジネスロジック (LogonBLogic.java)
 - BLogic インターフェース実装クラスの記述例を示す。ビジネスロジックの実行に関する詳細は、『WH-01 ビジネスロジック実行機能』を参照のこと。
 - 記述は省略するが、`execute()`メソッド内から呼び出されるメソッド `isAuthenticated()`、`getUserName()` は DB アクセスを行なう必要がある。DB アクセス方法については、『CB-01 データベースアクセス機能』を参照のこと。

```
public class LogonBLogic implements BLogic<LogonBLogicParam> {
    .....
    public BLogicResult execute(LogonBLogicParam params) {
        // 入力値オブジェクトから値を取得する。
        String uid = params.getUId();
        String pwd = params.getPwd();

        // 出力値を生成。
        BLogicResult bResult = new BLogicResult();

        // ログイン認証を行う。
        if(isAuthenticated(uid, pwd)) {
            // DBから値を取得。
            String userName = getUserName(uid);

            // ユーザバリューオブジェクトの生成。
            SampleUserValueObject uvo = new SampleUserValueObject();
        }
    }
}
```

blogic-io.xml の `blogic-params` 要素の `bean-name` 属性に指定したオブジェクトを入力値オブジェクトとして指定する。入力値がない場合は指定は不要。

blogic-io.xml の `set-property` 要素の `blogic-property` 属性で指定した名前を入力オブジェクトから値を取得できる。

(続く)

```
// UVOに値を設定
uvo.setId(uid);
uvo.setName(userName);

// 結果オブジェクト。
LogonBLogicResult result = new LogonBLogicResult();
result.setUvo(uvo);

// 出力値に結果オブジェクト (POJO) を設定。
bResult.setResultObject(result);

// 出力値にビジネスロジックの処理結果を表す文字列を設定。
bResult.setResultString("success");
} else {
// メッセージ生成。
BLogicMessages messages = new BLogicMessages();
messages.add("message", new BLogicMessage("blogic.message"));

// 出力値にメッセージを設定。
bResult.setMessages(messages);

// 出力値にビジネスロジックの処理結果を表す文字列を設定。
bResult.setResultString("failure");
}
return result;
}

private boolean isAuthenticated(String userId, String password) {
// DBにアクセスし、ユーザが存在するかの判定を行なう。
.....
}

private String getUsername(String userId) {
// DBにアクセスし、ユーザ名を取得する。
.....
}
}
```

bllogic-io.xml の set-property 要素の bllogic-property 属性で指定した名前で結果オブジェクトに値を設定する。

ビジネスロジックの実行結果 (POJO) を格納する。マッパークラスで処理され、プレゼンテーション層に反映される。

ここに設定した文字列から、Struts 設定ファイルの遷移先論理名を特定し、次の遷移先を決定する。

画面上などにメッセージを出力したい場合、BLogicMessages を作成して格納する。。

以上により、/logon.do のリクエストが発生した時、認証に必要な情報をアクションフォームから取得し、認証が成功した場合は、結果がセッションに格納される。

◆ 拡張ポイント

ビジネスロジックの入出力情報のマッピングルールのカスタマイズの際に、**BLogicMapper** を差し替えることが可能である。マッパークラスは、**AbstractBLogicMapper** 実装クラスを新規に作成するか、デフォルトで提供する **BLogicMapper** を拡張して作成する。**AbstractBLogicMapper**、**BLogicMapper** の拡張方法は、各クラスの Javadoc を参照のこと。

Struts 設定ファイル (**struts-config.xml**) において、**<set-property>** 要素の **mapperClass** プロパティの値に、使用したいマッパークラスの完全修飾名を定義することで、ビジネスロジック実行時にそのマッパークラスが処理されるようになる。

● Struts 設定ファイル (struts-config.xml)

```
<struts-config>
.....
<!-- BLogic-IO PlugIn -->
<plug-in className="jp.co.nttdata.terasoluna.fw.web.struts.plugins.BLogicIOPlugIn">
  <set-property property="resources" value="/WEB-INF/blogic-io.xml"/>
  <set-property property="digesterRules" value="/WEB-INF/blogic-io-rules.xml"/>
  <set-property property="mapperClass"
                value="jp.terasoluna.example.web.plugin.BLogicMapperEx"/>
</plug-in>
.....
</struts-config>
```

“mapperClass”プロパティの値として、
拡張マッパークラスの完全修飾名を指定する。

※ 注意点として、ビジネスロジック入出力情報設定ファイルの対象は自モジュールのみとなる。

また、このプロパティ設定を省略した場合、**jp.terasoluna.fw.service.thin.BLogicMapper** がデフォルトで設定される。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.service.thin.AbstractBLogicMapper	ビジネスロジック入出力情報を保持した BLogicResources をもとに、プレゼンテーション層のオブジェクトと、ビジネスロジック間のデータのマッピングを行なう機能を集約した抽象クラス。
2	jp.terasoluna.fw.service.thin.BLogicMapper	TERASOLUNA Server Framework for Java (Web 版)が提供するデフォルトの AbstractBLogicMapper 実装クラス。
3	jp.terasoluna.fw.web.struts.plugins.BLogicIOPlugin	ビジネスロジックの実行に必要となる、入出力情報を取得する。サーブレット起動時に実行される Struts プラグイン。
4	jp.terasoluna.fw.service.thin.BLogicResources	1 モジュールあたりのビジネスロジック入出力情報を格納する。ビジネスロジック入出力情報定義ファイル (blogic-io.xml) の<blogic-io>要素内の情報に相当する。
5	jp.terasoluna.fw.service.thin.BLogicIO	1 アクションあたりのビジネスロジック入出力情報を格納する。blogic-io.xml の<action>要素内の情報に相当する。
6	jp.terasoluna.fw.service.thin.BLogicProperty	blogic-io.xml の<blogic-params>または<blogic-result>要素内の<set-property>要素に相当する。

■ 関連機能

- 『WB-01 ユーザ情報保持機能』
- 『WH-01 ビジネスロジック実行機能』
- 『CB-01 データベースアクセス機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC23 ビジネスロジック入出力」
 - ◇ /webapps/blogicio/*
 - ◇ /webapps/WEB-INF/blogicio/*
 - ◇ jp.terasoluna.thin.functionsample.blogicio.*
- TERASOLUNA Server Framework for Java (Web 版) チュートリアル
 - 「2.4 ログオン」
 - /webapps/WEB-INF/blogic-io.xml

■ 備考

- なし

WI-01 一覧表示機能

■ 概要

◆ 機能概要

- Struts が提供する<logic:iterate>要素を使用して表示された一覧表に対して、ページリンク機能<ts:pageLinks>要素を使用することができる。
- 一覧表示機能、ページリンク機能ともに一画面に複数記述することができる。

◆ 概念図



◆ 解説

- <logic:iterate>
Struts の一覧表示機能<logic:iterate>要素を使用する。属性にて指定された Bean から一覧情報を取得して、取得した一覧情報分のループを回す。指定する Bean は、Collection、ArrayList、Vector、Enumeration、Iterator、Map、HashMap、Hashtable、TreeMap、配列などである。繰り返し項目（HTML テーブル内の<TR>~</TR>など）を囲むように記述する。詳細な使用方法は Struts を参照のこと。
- <ts:pageLinks>
<logic:iterate>要素によって定義された一覧のページ遷移のリンクを表示する。ページリンク機能を使用する場合は、アクションフォームに以下のプロパティを用意する必要がある。
 - 表示行数を保持するプロパティ
 - 表示開始インデックスを保持するプロパティ
 - 一覧情報の全件数を保持するプロパティ

■ 使用方法

◆ コーディングポイント

- 一覧情報を常にデータベースから取得する使用例

- JSP

- ◇ 属性の詳細は属性一覧を参照のこと。

```

<ts:pageLinks action="/list"
  rowProperty="row"
  totalProperty="totalCount"
  indexProperty="startIndex" />
<table border="1" frame="box">
  <logic:iterate id="userBean"
    name="userBeans" scope="request">
    . . . . .
  </logic:iterate>
</table>

```

一覧情報を取得するアクションを指定

name 属性を設定しない場合は、アクションフォームや Bean から取得をせずに直接リクエストやセッションから取得する。

リクエストに格納されている一覧情報をすべて表示する。

- Struts 設定ファイル

- ◇ 一覧表示に必要なアクションフォームのプロパティと、ページリンク押下時のアクションを Struts 設定ファイルに設定する。

```

<form-beans>
  <form-bean name="listForm"
    type="jp.terasoluna.fw.web.struts.form.DynaValidatorActionFormEx">
    <form-property name="row"
      type="java.lang.String" initial="10"/>
    <form-property name="startIndex"
      type="java.lang.String" initial="0"/>
  </form-bean>
</form-beans>

<action path="/list"
  name="listForm" scope="request">
  <forward name="success" path="/listSCR.do"/>
</action>

```

リクエストパラメータの表示行数と開始インデックスを取得し、ビジネスロジックに渡すためのアクションフォーム項目

ページ単位に一覧情報を取得するビジネスロジックを呼び出すアクションマッピング

- ◇ row、startIndex、totalCount は、アクションフォームではなく、リクエスト、セッションに持たせて使用することも可能である。name 属性を指定しない場合は、指定されたスコープに存在するプロパティ値を取得する。

➤ Bean 定義ファイル

- ◇ Bean 定義ファイルでは、起動するアクションの選択および、アクションから起動するビジネスロジックの設定を行う。

```
<bean name="/list" scope="prototype"
  class="jp.terasoluna.sample.web.action.ListAction">
  <property name="listService">
    <ref local="listService"/>
  </property>
</bean>

<bean id="listService" scope="prototype"
  class="jp.terasoluna.sample.service.ListService">
  <property name="dao"><ref bean="queryDAO"/></property>
</bean>
```

アクションに起動するビジネスロジックを設定する。

➤ BLogic 入出力設定ファイル

- ◇ アクションへ渡す値、およびアクションからの戻り値の設定を行う。

```
<action path="/list">
  <blogic-params bean-name="jp.terasoluna.sample.bean.ListBean">
    <set-property property="startIndex" source="form" />
    <set-property property="row" source="form" />
  </blogic-params>
  <blogic-result>
    <set-property property="userBeans" dest="request" />
    <set-property property="startIndex" dest="request" />
    <set-property property="row" dest="request" />
    <set-property property="totalCount" dest="request" />
  </blogic-result>
</action>
```

アクションフォームから開始インデックスと表示行数を JavaBean に格納して、アクションに渡す。

アクションから返却された BLogicResult に格納された結果オブジェクトをセッション、リクエスト、フォームのいずれかに格納する。

ここで、ページリンク機能に必要な開始インデックス、表示行数、総件数をリクエストに設定した場合、ページリンク機能の name 属性には何も設定せずに、直接リクエストから取得するようにプロパティ属性にのみ設定する。

➤ アクション

- ◇ ページリンクを押下された際に起動するアクションである。このアクションは `AbstractBLogicAction` を継承している。

```
public class ListAction extends AbstractBLogicAction<ListBean> {
    private ListService
        listService = null;
    .....
    public BLogicResult doExecuteBLogic(ListBean listBean)
        throws Exception {
        //ビジネスロジックの実行、結果の取得
        ResultBean resultBean
            = listService.getUserList(listBean);
        resultBean.setStartIndex(listBean.getStartIndex());
        resultBean.setRow(listBean.getRow());

        //BLogicResultの生成、結果の設定
        BLogicResult result = new BLogicResult();
        result.setResultString("success");
        result.setResultObject(resultBean);
        return result;
    }
}
```

Bean 定義ファイルにてアクションにビジネスロジックを設定する。

ビジネスロジックを実行して、一覧情報を取得する。

結果オブジェクトを BLogicResult に格納する。

➤ ビジネスロジック

- ◇ ページリンクを押下された際に起動するアクションから呼ばれるビジネスロジックである。

```
public ResultBean getUserList(ListBean listBean) {
    //開始インデックスと表示行数の取得
    int startIndex = listBean.getStartIndex();
    int row = listBean.getRow();
    //総件数取得
    Integer totalCount = dao.executeForObject(
        "getUserCount", null, Integer.class);
    //一覧情報を画面表示分のみ取得する。
    UserBean[] beans = dao.executeForObjectArray("getUserList",
        null, UserBean.class, startIndex, row);
    ResultBean resultBean = new ResultBean();
    resultBean.setTotalCount(totalCount.intValue());
    resultBean.setUserBeans(beans);
    return resultBean;
}
```

開始インデックスと表示行数を取得して、DAO の引数に設定する。

取得した総件数と一覧情報を Bean に格納してアクションへ返却する。

- 一覧情報をアクションフォームから取得する使用例

- JSP

- ◇ 属性の詳細は属性一覧を参照のこと。

```

<ts:pageLinks action="/listSCR"
  name="_listForm" rowProperty="row"
  totalProperty="totalCount"
  indexProperty="startIndex"
  scope="session" />
<table border="1" frame="box">
  <bean:define id="startIndex"
    name="_listForm" property="startIndex"
    scope="session" type="java.lang.String" />
  <logic:iterate id="userBean" scope="session"
    name="_listForm" property="userBeans"
    length="10" offset="<%=startIndex%>" />
  . . . . .
</logic:iterate>
</table>

```

画面を表示するだけのアクションを指定

アクションフォームに定義したプロパティを指定する。

一覧情報を offset 属性で指定したインデックスから、length 属性で指定した件数だけ表示する。offset 属性には、事前にフォームから取り出した開始インデックスの値を設定する。

- Struts 設定ファイル

- ◇ アクションフォームを使用する一覧表示の場合、一覧情報および関連するプロパティをセッションに格納する必要がある。

```

<form-bean name="_listForm"
  type="jp.terasoluna.fw.web.struts.form.DynaValidatorActionFormEx">
  <!-- 一覧表示用 -->
  <form-property name="userBeans"
    type="jp.terasoluna.sample.bean.UserBean[]" />
  <form-property name="row"
    type="java.lang.String" initial="10"/>
  <form-property name="startIndex"
    type="java.lang.String" initial="0"/>
  <form-property name="totalCount"
    type="java.lang.String"/>
</form-bean>
<action path="/list"
  name="_listForm" scope="session">
  <forward name="success" path="/listSCR.do" />
</action>
<action path="/listSCR"
  name="_listForm" scope="session"
  parameter="/list.jsp">

```

表示行数

表示開始インデックス

一覧情報全件数

すべての一覧情報と総件数を取得するアクション

ページリンク機能にて呼び出される画面表示アクション。name 属性は対応するアクションフォームを指定、scope 属性は“session”にする必要がある。

➤ Bean 定義ファイル

- ◇ Bean 定義ファイルでは、起動するアクションの選択および、アクションから起動するビジネスロジックの設定を行う。

```
<bean name="/list" scope="prototype"
  class="jp.terasoluna.sample.web.action.ListAction">
  <property name="listService">
    <ref local="listService"/>
  </property>
</bean>
```

すべての一覧情報と総件数を取得するアクションを指定する。一覧画面を表示する最初にしか呼び出されない。

```
<bean name="/listSCR" scope="prototype"
  class="jp.terasoluna.fw.web.struts.actions.ForwardAction"/>
```

一覧画面を再表示するだけのアクション ForwardAction を指定する。

➤ BLogic 入出力設定ファイル

- ◇ アクションへ渡す値、およびアクションからの戻り値の設定を行う。

```
<action path="/list">
  <blogic-params bean-name="java.util.HashMap">
  </blogic-params>
  <blogic-result>
    <set-property property="userBeans" dest="form" />
    <set-property property="totalCount" dest="form" />
  </blogic-result>
</action>
```

例では、入力値がないためプロパティがないが、検索条件などがある場合は記述する。

アクションから返却された BLogicResult に格納された結果オブジェクトをフォームに格納する。

➤ アクション

- ◇ ページリンクを押下された際に起動するアクションである。このアクションは `AbstractBLogicAction` を継承している。

```
public class ListAction extends AbstractBLogicAction {  
    private ListService  
        listService = null;  
    .....  
    public BLogicResult doExecuteBLogic(Object obj)  
        throws Exception {  
  
        //ビジネスロジックの実行、結果の取得  
        ResultBean resultBean = listService.getUserList();  
  
        //BLogicResultの生成、結果の設定  
        BLogicResult result = new BLogicResult();  
        result.setResultString("success");  
        result.setResultObject(resultBean);  
        return result;  
    }  
}
```

Bean 定義ファイルにてアクションにビジネスロジックを設定する。

ビジネスロジックを実行して、一覧情報を取得する。

結果オブジェクトを BLogicResult に格納する。

➤ ビジネスロジック

- ◇ ページリンクを押下された際に起動するアクションから呼ばれるビジネスロジックである。

```
public ResultBean getUserList() {  
    //一覧情報を取得する。  
    UserBean[] beans = dao.executeForObjectArray("getUserList",  
        null, UserBean.class);  
    ResultBean resultBean = new ResultBean();  
    resultBean.setTotalCount(beans.length);  
    resultBean.setUserBeans(beans);  
    return resultBean;  
}
```

取得した総件数と一覧情報を Bean に格納してアクションへ返却する。

- サブミットを行いたい場合
ページリンク機能はデフォルトではリンクによるページ遷移を行うが、submit 属性を true に設定することで JavaScript によるサブミット処理を行うことができる。また、サブミットにてページ遷移する場合は、<ts:form>要素を記述すること。

➤ JSP

- ◇ 属性の詳細は属性一覧を参照のこと。

```
<ts:pageLinks action="/list" submit="true" rowProperty="row"
              totalProperty="totalCount" indexProperty="startIndex" />
```

- DispatchAction を使用する場合

サブミットの処理を振り分けるために DispatchAction を使用する場合、ページリンク機能では、submit 属性と forward 属性と event 属性を使用する。

➤ JSP

- ◇ submit 属性および forward 属性を true に設定すると、event 属性に設定された値の Hidden 項目を出力する。ページリンクが押下されたとき JavaScript にて、その Hidden 項目に値 “forward_pageLinks” を設定する。
- ◇ event 属性のデフォルト値は “event” である。
- ◇ 属性の詳細は属性一覧を参照のこと。

```
<ts:form action="/listDSP">
  <ts:pageLinks action="/list" name="dynaFormBean" rowProperty="row"
                totalProperty="totalCount" indexProperty="startIndex"
                submit="true" forward="true" event="next"
  .....
</ts:form>
```

➤ Struts 設定ファイル

- ◇ ページ遷移のフォワード名は “pageLinks” を設定する。

```
<action path="/listDSP"
        name="dynaFormBean" scope="request">
  <forward name="pageLinks" path="/list.do" />
  <forward name="delete" path="/delete.do" />
  <forward name="default" path="/list.do" />
</action>
```

➤ Bean 定義ファイル

- ◇ DispatchAction の event プロパティに<ts:pageLinks>要素の event 属性の値と同じ値を指定する。DispatchAction の event プロパティのデフォルト値も “event” であるため、ページリンク機能の event 属性の値を指定していない場合は、不要である。

```
<bean name="/listDSP" scope="prototype"
      class="jp.terasoluna.fw.web.struts.actions.DispatchAction">
  <property name="event"><value>next</value></property>
</bean>
```

- 指定範囲リセットを行う場合

リセット機能の指定範囲リセットを行う場合、「startIndex」と「endIndex」がリクエストパラメータに必要となる。ページ遷移リンク機能では、resetIndex 属性を true に設定すると、「startIndex」と「endIndex」が Hidden 項目として出力される。

- JSP

```
<ts:pageLinks action="/list" rowProperty="row"
              totalProperty="totalCount" indexProperty="startIndex"
              resetIndex="true"/>
```

- HTML の出力結果

```
<input type="hidden" name="startIndex" value="0"/>
<input type="hidden" name="endIndex" value="9"/>
```

指定範囲リセット機能の詳細は『WB-04 フォームプロパティリセット機能』を参照のこと。

- id 属性の指定による出力場所の自由化

通常、ページリンク機能はリンクを出力したい場所に<ts:pageLinks>要素を記述するが、id 属性に文字列を指定すると、その文字列をキーにページコンテキストにページリンクの出力を行う。ページコンテキストに保存されたページリンクは、<bean:write>要素などで出力させることができるため、<ts:pageLinks>要素以降であれば好きな場所に出力させることができる。

注意点として、<bean:write>要素などで出力する際は、サニタイジング処理は行わないこと。<bean:write>要素の場合、filter 属性を false にする必要がある。

- JSP

```
<ts:pageLinks id="reservePageLinks" action="/list"
              rowProperty="row" totalProperty="totalCount"
              indexProperty="startIndex" resetIndex="true"/>
<bean:write name="reservePageLinks" filter="false"/>
```

- 現在ページ、総ページ数、総件数の出力

ページリンク機能では、`currentPageIndex` 属性および `totalPageCount` 属性に指定された文字列をキーにして、現在ページと総ページ数をページコンテキストに設定している。また、これらの属性にはデフォルト値が設定されていて、指定しない場合でもデフォルト値を使用して、ページコンテキストへ設定をしている。ページコンテキストから画面への出力は`<bean:write>`要素などを使用する。総件数はページリンク機能にて必要なため、フォームまたはセッションやリクエストにすでに保持されているはずである。画面への出力は`<bean:write>`要素を使用する。

- JSP（属性指定をしない場合）

```
<ts:pageLinks action="/list" rowProperty="row"
    totalProperty="totalCount" indexProperty="startIndex"/>
現在は<bean:write name="currentPageIndex" />ページです。
全部で<bean:write name="totalPageCount" />ページあります。
全部で<bean:write name="totalCount" />件あります。
```

- JSP（属性指定の場合）

```
<ts:pageLinks action="/list" rowProperty="row"
    totalProperty="totalCount" indexProperty="startIndex"
    currentPageIndex="nowPage" totalPageCount="totalPage"/>
<bean:write name="nowPage" />
<bean:write name="totalPage" />
```

- 出力リンクの変更

ページリンク機能では、出力するリンクのフォーマットをプロパティファイルで変更することができる。以下に設定することのできるプロパティを記述する。

キー	デフォルト	概要
<code>pageLinks.prev<数値>.char</code>	-	現在のページより前ページに遷移するリンクを出力する。「<数値>」部分に記述した数の前ページへ遷移する。10 と指定したら、現在のページから 10 ページ前に遷移する。「<数値>」部分を変更することで複数記述することができる。
<code>pageLinks.next<数値>.char</code>	-	現在のページより次ページに遷移するリンクを出力する。「<数値>」部分に記述した数の次ページへ遷移する。10 と指定したら、現在のページから 10 ページ次に遷移する。「<数値>」部分を変更することで複数記述することができる。
<code>pageLinks.maxDirectLinkCount</code>	10	前ページと次ページの間に表示するページ番号リンクの数を指定する。このキーに「5」を設定し、一覧のページが 10 ページで現在ページが 5 ページの場合、画面には「 3 4 5 6 7 」のページ番号リンクが出力される。

➤ プロパティ設定例

```
pageLinks.prev10.char=10ページ前へ
pageLinks.prev5.char=5ページ前へ
pageLinks.prev1.char=前ページ
pageLinks.next1.char=次ページ
pageLinks.next5.char=5ページ次へ
pageLinks.next10.char=10ページ次へ
pageLinks.maxDirectLinkCount=5
```

➤ 上記の設定での出力例

- ◇ プロパティファイルを上記のように設定した一覧画面の 13 ページを表示した場合の出力例

```
10ページ前へ 5ページ前へ 前ページ 11 12 13 14 15 次ページ 5ページ次へ 10ページ次へ
```

● 画像リンクの使用

ページリンク機能では、以下のようにプロパティファイルを記述することでリンク部分を画像リンクにすることができる。

➤ プロパティ設定例

- なお、注意点として、画像リンクを使用する場合は、「border="0"」を記述しないと、画像の周りにリンク枠が出力されるため注意すること。

```
pageLinks.prev5.char=
pageLinks.prev1.char=
pageLinks.next1.char=
pageLinks.next5.char=
```

◆ 拡張ポイント

なし。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.struts.taglib. PageLinksTag	ページ単位にページ遷移するリンクを提供するカスタムタグ

■ リファレンス

◆ タグ属性一覧

属性	必須	概要
id	-	この属性に文字列が指定された場合、ページリンクの出力先を画面ではなくページコンテキストに保存する。この属性は保存するキーとなる。
action	△	一覧表示画面の表示を行うアクションパス名を指定する。 submit 属性が false の場合は必須属性となる。
name	-	表示行数、開始行インデックス、一覧情報全行数を取得する Bean を指定する。指定しない場合は表示行数、開始行インデックス、一覧情報全行数はスコープから直接取得する。
rowProperty	○	表示行数のプロパティを指定する。
indexProperty	○	開始行インデックスのプロパティを指定する。
totalProperty	○	全行数のプロパティを指定する。
scope	-	name 属性、3つのプロパティ属性で指定した Bean を取得するスコープを指定する。
submit	-	リンクではなく、サブミットを行う場合は true を指定する。デフォルトは false。
forward	-	submit 属性が true のときに有効となる属性で、TERASOLUNA Server Framework for Java (Web 版)の DispatcherServlet を使用してフォワードによる振り分けを行う場合に使用する。true を指定すると event 属性に設定された値の Hidden タグを出力する。また、サブミット時に、その Hidden タグの value 属性に“forward_pageLinks”を設定する。デフォルトは false。
event	-	forward 属性が true のときに有効となる属性で、TERASOLUNA Server Framework for Java (Web 版)の DispatcherServlet を使用してフォワードによる振り分けを行う場合に使用する。この属性に指定した名前の Hidden タグが生成される。デフォルトは“event”となる。
resetIndex	-	指定範囲リセットを行うための startIndex と endIndex の Hidden タグを出力する。デフォルトは false。
currentPageIndex	-	対応する一覧の現在ページ数をページコンテキストに保存する際のキーとなる。デフォルトは“currentPageIndex”となる。
totalPageCount	-	対応する一覧の総ページ数をページコンテキストに保存する際のキーとなる。デフォルトは“totalPageCount”となる。

■ 関連機能

『WK-09 一覧表示関連機能』

■ 使用例

- Terasoluna Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC24 一覧表示」
 - ◇ /webapps/pagelink/*
 - ◇ /webapps/WEB-INF/pagelink/*
 - ◇ jp.terasoluna.thin.functionsample.pagelink.*
- Terasoluna Server Framework for Java (Web 版) チュートリアル
 - 「2.5 一覧表示」
 - 一覧表示画面

■ 備考

なし。

WJ-01～WK-09 画面表示機能

■ 概要

◆ 機能概要

- 画面表示（カスタムタグ）に関する機能である。
- TERASOLUNA Server Framework for Java (Web 版)が提供するカスタムタグには以下の二種類がある。
 - TERASOLUNA Server Framework for Java (Web 版)独自のカスタムタグ
 - Struts 提供のタグを拡張したカスタムタグ

◆ 提供カスタムタグ一覧

- TERASOLUNA Server Framework for Java (Web 版)独自のカスタムタグ

機能番号	機能名	タグ	機能概要
WJ-01	アクセス権限チェック機能	<t:ifAuthorized> <t:ifAuthorizedBlock>	ユーザの権限によって表示/非表示を切り替える。
WJ-02	サーバ閉塞チェック機能	<t:ifPreBlockade>	サーバが閉塞しているか否かで表示/非表示を切り替える。
WJ-03	カレンダー入力機能	<t:inputCalendar>	年月日をカレンダー画面から入力する機能。
WJ-04	文字列表示機能	<t:write>	指定した bean プロパティの値を変換し、表示する。
WJ-05	日付変換機能	<t:date>	指定された形式に従って日付・時刻をフォーマットする。
WJ-06	和暦日付変換機能	<t:jdate>	日付時刻データを和暦としてフォーマットする。
WJ-07	Decimal 表示機能	<t:decimal>	符号、および小数点付き数値をフォーマットして出力、あるいはスクリプティング変数として定義する。
WJ-08	トリム機能	<t:rtrim> <t:ltrim> <t:trim>	指定された文字列のスペースを削除する。
WJ-09	文字列切り取り機能	<t:left>	文字列の左端から指定された文字数分の文字列を切り出す。
WJ-10	コードリスト定義機能	<t:defineCodeList>	読み込み済みのコードリストを、jsp 内で利用する。
WJ-11	コードリスト件数出力機能	<t:writeCodeCount>	コードリストの要素数を出力する。
WJ-12	指定コードリスト値表	<t:writeCodeValue>	指定したコードリストの値を画面に表示する。

	示機能		
--	-----	--	--

● Struts 提供のタグを拡張したカスタムタグ

機能番号	機能名	タグ	機能概要
WK-01	スタイルクラス切り替え機能	<ts:changeStyleClass>	エラー時に、スタイルシートのクラスを切り替える。
WK-02	メッセージ表示機能	<ts:errors> <ts:messages>	メッセージまたはエラーメッセージ情報を表示する。
WK-03	キャッシュ避け form タグ機能	<ts:form>	アクション URL にキャッシュ避け用ランダム ID を追加する。
WK-04	キャッシュ避けリンク機能	<ts:link>	アクション URL にキャッシュ避け用ランダム ID を追加する。
WK-05	フォームターゲット指定機能	<ts:submit>	フォームのターゲットを指定する。
WK-06	メッセージポップアップ機能	<ts:messagesPopup> <ts:body>	メッセージまたはエラーメッセージをポップアップ画面に表示する。
WK-07	エラーメッセージチェック機能	<ts:ifErrors> <ts:ifNotErrors>	リクエスト、またはセッションにエラー情報が設定されているかどうかを判別して表示/非表示を切り替える。
WK-08	クライアントチェック拡張機能	<ts:javascript>	クライアントでの入力チェック時に検証に使用する値を javascript の変数として出力する。
WK-09	一覧表示関連機能	<logic:iterate> <ts:pageLinks>	Struts の一覧表示機能<logic:iterate>要素を使用する。ページングのためのタグを提供する。

■ WJ-01 アクセス権限チェック機能

◆ 概要

現在のログオンユーザが指定されたパスへのアクセス権限をもっているかどうかを判別して表示/非表示を切り替える。

◆ 解説

- `<t:ifAuthorized>`
path 属性で指定されたパスに対してアクセス権がある場合にのみ、タグのボディ部分を評価する。アクセス権のチェックは、AuthorizationController が行う。AuthorizationController に関しては、『WA-02 アクセス権限チェック機能』を参照のこと。
- `<t:ifAuthorizedBlock>`
`<t:ifAuthorized>`要素の結果を blockId 毎に制御する為の要素で、blockId で `<t:ifAuthorized>`要素と紐付けられ、ボディ内を表示するかどうかを判定する。この要素を入れ子状にすることで、アクセス権限毎の制御を柔軟に行なう事が可能になる。
入れ子状にする場合は、親要素の blockId 属性と子要素の parentBlockId 属性が紐付けられ、ボディ内を表示するかどうかを判定する。

◆ 使用方法例

- `<t:ifAuthorized>`要素の使用例

```
<t:ifAuthorized path="/pathToSomewhere">
```

パス「/pathToSomewhere」へのアクセス権限がある場合、出力される。

```
</t:ifAuthorized>
```

- <t:ifAuthorizedBlock>要素の使用例

```
<t:ifAuthorizedBlock blockId="ABC" >
  ボディ内の blockId で紐付けられた
  IfAuthorizedBlockTag が表示される場合のみ表示される。
  <t:ifAuthorizedBlock blockId="EFG" parentBlockId="ABC" >
    ボディ内の blockId で紐付けられた IfAuthorizedTag
    が表示される場合のみ表示される。
    <t:ifAuthorized path="/sample1/test.do" blockId="EFG" >
      パス「/sample1/test.do」へのアクセス権限がある場合、出力される。
    </t:ifAuthorized>
  </t:ifAuthorizedBlock>
</t:ifAuthorizedBlock>
```

◆ タグ属性一覧

- <t:ifAuthorized>要素の属性一覧

属性	必須	概要
Path	○	対象となる path を指定する。
blockId	-	このタグの親となる<htmlx:IfAuthorizedBlock>要素と紐付ける為の blockId を指定する。

- <t:ifAuthorizedBlock>要素の属性一覧

属性	必須	概要
blockId	○	対象となる blockId を指定する。
parentBlockId	-	このタグの親となる<htmlx:IfAuthorizedBlock>要素と紐付ける為の blockId を指定する。

◆ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC05 アクセス権限チェック」
 - ◇ /webapps/authorization/*
 - ◇ /webapps/WEB-INF/authorization/*
 - ◇ jp.terasoluna.thin.functionsample.authorization.*

■ WJ-02 サーバ閉塞チェック機能

◆ 概要

サーバが予閉塞または閉塞状態かどうかを判別して表示/非表示を切り替える。

◆ 解説

- `<t:ifPreBlockade>`
サーバが閉塞状態又は予閉塞状態の場合にのみ、タグのボディ部分を出力する。
サーバ閉塞のチェックは、`ServerBlockageController` が行う。
`ServerBlockageController` に関しては、『WA-02 サーバ閉塞チェック機能』を参照のこと。

◆ 使用方法例

```
<t:ifPreBlockade>  
... // サーバが閉塞状態又は予閉塞状態の場合にのみの表示項目等  
</t:ifPreBlockade>
```

◆ タグ属性一覧

なし。

◆ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC06 サーバ閉塞チェック」
 - ◇ `/webapps/serverblockage/*`
 - ◇ `/webapps/WEB-INF/serverblockage/*`
 - ◇ `jp.terasoluna.thin.functionsample.serverblockage.*`

■ WJ-03 カレンダー入力機能

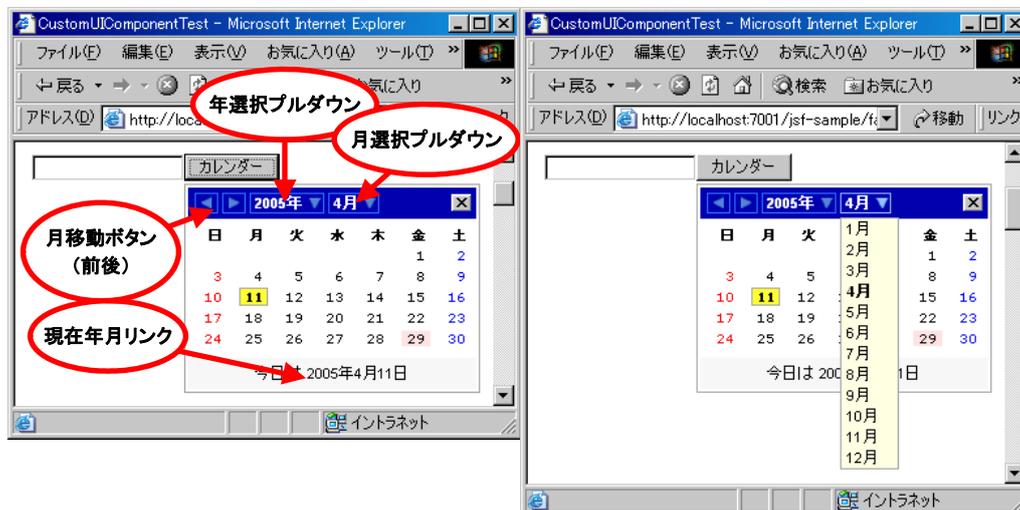
◆ 概要

指定されたテキストフィールドに対して、カレンダー入力機能を提供する。

◆ 解説

- <t:inputCalendar>

JavaScript を使用したカレンダー画面を表示し選択した日付を、指定された入力フィールドに入力する。



◆ 機能詳細

- 設定ファイル
国際化対応のため、メッセージリソースファイルを設定ファイルとして利用する
メッセージリソースファイルはクラスパス直下に「calendar.properties」というファイル名で作成する。
- 休日定義
メッセージリソースファイルに以下のように記述することでカレンダーの休日を指定することができる。メッセージリソースキーは、「calendar.holiday.」部分を固定として、その後に、「1」から連番を振ることとする。パラメータは、「年」「月」「日」「休日概要」を「, (カンマ)」で区切って記述することとする。毎年同じ日付の休日定義は「年」を「0」と指定することで毎年と認識する。

```
calendar.holiday.1=0,1,1,元旦  
calendar.holiday.2=0,2,11,建国記念日  
calendar.holiday.3=0,4,29,みどりの日  
calendar.holiday.4=2005,1,10,成人の日  
calendar.holiday.5=2005,3,20,春分の日  
calendar.holiday.6=2005,9,19,敬老の日  
calendar.holiday.7=2005,9,23,秋分の日  
calendar.holiday.8=2005,10,10,体育の日
```

- ボタン表示文字列変更

メッセージリソースファイルに以下のように記述することでカレンダーを表示するボタンの文字列を変更することができる。

メッセージリソースキーは、「calendar.button.string」固定とする。デフォルトは「Calendar」となる。

```
calendar.button.string=カレンダー
```

- スタイルプレフィックス変更

メッセージリソースファイルに以下のように記述することでカレンダーにて使用するスタイルシートのプレフィックス、および画像ファイルのプレフィックスを変更することができる。

メッセージリソースキーは、「calendar.style.themeprefix」固定とする。デフォルトは「BlueStyle」となる。

```
calendar.style.themeprefix=WhiteStyle
```

- 現在日付表示文字列変更

メッセージリソースファイルに以下のように記述することでカレンダーの下部に表示される現在日付に付与する文字列を変更することができる。

メッセージリソースキーは、「calendar.today.string」固定とする。デフォルトは「Today is」となる。

```
calendar.today.string=Today is
```

- カレンダー画像保存場所変更

メッセージリソースファイルに以下のように記述することでカレンダー入力機能にて使用する画像の保存場所を変更することができる。

最後は「/」で終わる必要がある。画像の保存場所は変更可能だが、画像ファイルの名前は変更することができない。

メッセージリソースキーは、「calendar.img.dir」固定とする。デフォルトは「img/calendar/」となる。

```
calendar.img.dir=image/
```

- スタイルシート保存場所変更

メッセージリソースファイルに以下のように記述することでカレンダー入力機能にて使用するスタイルシートの保存場所を変更することができる。最後は「/」

で終わる必要がある。この機能で使用するスタイルシートのファイル名は、「<プレフィックス>+InputCalendar.css」である。

メッセージリソースキーは、「calendar.stylesheet.dir」固定とする。デフォルトは「css/」となる。

```
calendar.stylesheet.dir=stylesheet/
```

- 外部 JavaScript ファイル保存場所変更

メッセージリソースファイルに以下のように記述することでカレンダー入力機能にて使用する外部 JavaScript の保存場所を変更することができる。最後は「/」で終わる必要がある。この機能で使用する JavaScript のファイル名は、「InputCalendar.js」である。

メッセージリソースキーは、「calendar.javascript.dir」固定とする。デフォルトは「js/」となる。

```
calendar.javascript.dir=javascript/
```

◆ 使用方法例

```
<html:text name="dynamicForm" property="date1" size="15" />
<t:inputCalendar for="date1" format="yyyy/MM/dd" />
```

◆ タグ属性一覧

属性	必須	概要
for	○	選択した日付を入力する入力フィールドを指定する。
format	-	カレンダーのフォーマットを指定する。 指定できる日付形式は「y(年)」「M(月)」「d(日)」、区切文字としては「/」「-」「.」「半角スペース」のいずれかである。また、区切り文字は、一文字のみを使用すること。「yyyy/MM-dd」のように複数の区切り文字を使用することはできない。
formatKey	-	カレンダーのフォーマットをメッセージリソースから取得するためのキー値を指定する。

◆ 使用例

- Terasoluna Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC25 カレンダー入力」
 - ◇ /webapps/calendar/*
 - ◇ /webapps/WEB-INF/calendar/*

- Terasoluna Server Framework for Java (Web 版) チュートリアル：登録画面
 - 「2.7 登録」

■ WJ-04 文字列表示機能

◆ 概要

指定した bean プロパティの値を変換し、表示する。

◆ 解説

- `<t:write>`
指定した bean プロパティの値を取り出し、String として現在の `JspWriter` に与える。
また、属性により以下の変換を行う。
 - null もしくは空文字を “ ” に置換
 - 半角スペースを “ ” に置換
 - 改行コードを `
` に置換
 - 改行文字を無視

◆ 使用方法例

```
primaryFieldの内容  
「 あいうえお” + System.getProperty("line.separator") + ” かきくけこ”」  
  
<t:write name="htmlxForm"  
        property="primaryField" />  
  
↓出力結果  
&nbsp;あいうえお<br>&nbsp;かきくけこ
```

◆ タグ属性一覧

属性	必須	概要
Filter	-	この属性が true にセットされる場合、表現されたプロパティ値は HTML 内でセンシティブな文字のためにフィルターされる。そしてこのような全ての文字は、等価な文字で置き換えられる。デフォルトでは、フィルタリングが行われる。無効にするためには、この属性に明示的に false をセットする必要がある。
replaceNullToNbsp	-	この属性が true にセットされ、指定した bean プロパティの値が空文字及び、null の場合、 を出力する。無効にするためには、この属性に明示的に false をセットする必要がある。
replaceSpToNbsp	-	この属性が true にセットされ、指定した bean プロパティの値に 1 Byte コードのスペースが存在する場合、 に置換する。無効にするためには、この属性に明示的に false をセットする必要がある。
replaceLFtoBR	-	この属性が true にセットされる場合、指定した bean プロパティの値の改行コードもしくは復帰文字が に置換される。無効にするためには、この属性に明示的に false をセットする必要がある。
Ignore	-	この属性が true にセットされ、name と scope 属性で指定した bean が存在しない場合、なにもせずにリターンする。デフォルト値は false(このタグライブラリの中のほかのタグと矛盾しないように実行時例外がスローされる)。
Name	○	property(指定がある場合)によって指定した値を取り出すために、プロパティがアクセスされる bean の属性名を指定する。property が指定されない場合、この bean 自身の値が表現される。
property	-	name によって指定した bean 上でアクセスされるプロパティの名前を指定する。この値はシンプル、インデックス付き、またはネストされたプロパティ参照式になる。指定されない場合は、name によって識別された bean はそれ自身を表現する。指定したプロパティが null を戻す場合、何も表現されない。
Scope	-	name によって指定した bean を取り出すために検索された可変スコープを指定する。指定されない場合、PageContext.findAttribute()によって適用されたデフォルトのルールが適用される。
fillColumn	-	fillColumn によって指定された文字数で区切り、区切った終端に を付与する。文字数の数え方は半角でも、全角でも1つの文字とみなす。
addBR	-	この属性が true にセットされる場合、プロパティ値の末尾に を付与する。デフォルトは false。

◆ 使用例

- Terasoluna Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC26 文字列表示」
 - ◇ /webapps/write/*
 - ◇ /webapps/WEB-INF/write/*
 - ◇ jp.terasoluna.thin.functionsample.write.*

■ WJ-05 日付変換機能

◆ 概要

指定された形式に従って日付・時刻をフォーマットする。

◆ 解説

- <t:date>
pattern 属性で指定された出力形式の文字列を java.text.SimpleDateFormat クラスの時刻パターン文字列として解釈し、フォーマットする。時刻パターン文字列の詳細については、java.text.SimpleDateFormat クラスのドキュメントを参照のこと。

◆ 使用方法例

```
<t:date name="form0001"  
        property="field001"  
        pattern="yyyy/MM/dd hh:mm aaa"/>
```

↓上記の出力結果

```
2005/7/14 11:24 PM
```

◆ タグ属性一覧

属性	必須	概要
id	-	フォーマットされた文字列をレスポンスへ出力せずに、スクリプティング変数にセットする際に指定する。フォーマットされた文字列をスクリプティング変数にセットする場合には、filter 属性の指定に関わらず HTML 特殊文字はエスケープされない。
filter	-	フォーマットされた文字列を出力する際に、HTML 特殊文字をエスケープするかどうかを指定する。ただし、id 属性が指定されていた場合には、無視される。
ignore	-	name 属性で指定した bean が見つからなかったときに無視するかどうかを指定する。false を指定すると、bean が見つからなかったときに JspException が投げられる。
name	-	フォーマット対象の文字列をプロパティを持つ bean の名前。property 属性が指定されていなかったときは、name 属性で指定されたインスタンスがフォーマットの対象となる。この場合、そのインスタンス自身が java.util.Date 型であるか、あるいは java.lang.String 型(かつ yyyy/MM/dd hh:mm:ss の形式となっているもの)のどちらかである必要がある。value 属性が指定されていた場合は無視される。
property	-	name 属性で指定された bean においてアクセスされるプロパティの名前。value 属性が指定されていた場合には無視される。
scope	-	name 属性で指定された bean を検索する際のスコープ。
value	-	フォーマットする文字列。文字列は、"yyyy/MM/dd hh:mm:ss"の形式となっている必要がある。value 属性を指定した場合には、name 属性、および property 属性は無視される。
pattern	○	フォーマットする出力形式。pattern 属性で指定した出力形式は、DateFormatterTagBase クラスのサブクラスで解釈される。詳細は、サブクラスのドキュメントを参照のこと。

◆ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC27 日付変換」
 - ◇ /webapps/date/*
 - ◇ /webapps/WEB-INF/date/*
 - ◇ jp.terasoluna.thin.functionsample.date.*

■ WJ-06 和暦日付変換機能

◆ 概要

日付時刻データを和暦としてフォーマットする。

◆ 解説

- <t:jdate>
日付時刻のデータのフォーマットを行う際に、和暦の元号（“昭和”、“S” など）や、和暦の年（西暦の“2002”年ではなく、平成“14”年など）、および曜日の日本語表記（“月曜日”、“月”など）に変換する。

◆ 機能詳細

- 和暦の設定
和暦のデータは `DateUtil` クラス経由でプロパティファイルから取得する。以下の形式でプロパティファイルに設定する

```
wareki.gengo.ID.name=元号名  
wareki.gengo.ID.roman=元号のローマ字表記  
wareki.gengo.ID.startDate=元号法施行日（西暦:yyyy/MM/dd形式）
```

以下は一般的な設定例である。

```
wareki.gengo.0.name = 平成  
wareki.gengo.0.roman = H  
wareki.gengo.0.startDate = 1989/01/08  
wareki.gengo.1.name = 昭和  
wareki.gengo.1.roman = S  
wareki.gengo.1.startDate = 1926/12/25  
wareki.gengo.2.name = 大正  
wareki.gengo.2.roman = T  
wareki.gengo.2.startDate = 1912/07/30  
wareki.gengo.3.name = 明治  
wareki.gengo.3.roman = M  
wareki.gengo.3.startDate = 1868/09/04
```

◆ 使用方法例

```
<t:jdate name="form0001"
         property="field001"
         pattern="GGGGyy年MM月dd日(EEEE) hh時mm分ss秒"/>
```

↓上記の出力結果

平成17年07月14日(木曜日) 11時24分31秒

◆ タグ属性一覧

属性	必須	概要
id	-	フォーマットされた文字列をレスポンスへ出力せずに、スクリプティング変数にセットする際に指定する。フォーマットされた文字列をスクリプティング変数にセットする場合には、filter 属性の指定に関わらず HTML 特殊文字はエスケープされない。
filter	-	フォーマットされた文字列を出力する際に、HTML 特殊文字をエスケープするかどうかを指定する。ただし、id 属性が指定されていた場合には、無視される。
ignore	-	name 属性で指定した bean が見つからなかったときに無視するかどうかを指定する。false を指定すると、bean が見つからなかったときに JspException が投げられる。
name	-	フォーマット対象の文字列をプロパティに持つ bean の名前。property 属性が指定されていなかったときは、name 属性で指定されたインスタンスがフォーマットの対象となる。この場合、そのインスタンス自身が java.util.Date 型であるか、あるいは java.lang.String 型(かつ"yyyy/MM/dd hh:mm:ss"の形式となっているもの)のどちらかである必要がある。Value 属性が指定されていた場合は無視される。
property	-	name 属性で指定された bean においてアクセスされるプロパティの名前。value 属性が指定されていた場合には無視される。
scope	-	name 属性で指定された bean を検索する際のスコープ。
value	-	フォーマットする文字列。文字列は、yyyy/MM/dd hh:mm:ss の形式となっている必要がある。value 属性を指定した場合には、name 属性、および property 属性は無視される。
pattern	-	フォーマットする出力形式。pattern 属性で指定した出力形式は、DateFormatterTagBase クラスのサブクラスで解釈される。詳細は、JDateTag の JavaDoc を参照のこと。

◆ 使用例

- Terasoluna Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC28 和暦日付変換」
 - ◇ /webapps/wareki/*
 - ◇ /webapps/WEB-INF/wareki/*
 - ◇ jp.terasoluna.thin.functionsample.wareki.*

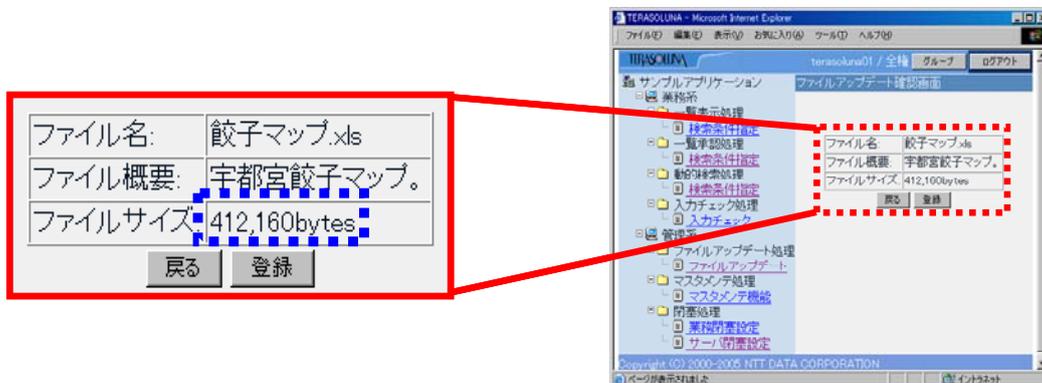
■ WJ-07 Decimal 表示機能

◆ 概要

数値をフォーマットして出力する。

◆ 解説

- <t:decimal>
符号、および小数点付き数値をフォーマットして出力、あるいはスクリプティン
グ変数として定義する。



◆ 使用方法例

```
<td nowrap>ファイルサイズ:</td>  
<td nowrap>  
  <t:decimal name="_fileForm" property="fileSize"  
    pattern="###,###bytes" />  
</td>
```

◆ タグ属性一覧

属性	必須	概要
id	-	フォーマットされた文字列をレスポンスへ出力せずに、スクリプティング変数にセットする際に指定する。フォーマットされた文字列をスクリプティング変数にセットする場合には、filter 属性の指定に関わらず HTML 特殊文字はエスケープされない。
filter	-	フォーマットされた文字列を出力する際に、HTML 特殊文字をエスケープするかどうかを指定する。ただし、id 属性が指定されていた場合には、無視される。
ignore	-	name 属性で指定した bean が見つからなかったときに無視するかどうかを指定する。false を指定すると、bean が見つからなかったときに JspException が投げられる。
name	-	フォーマット対象の文字列をプロパティを持つ bean の名前。property 属性が指定されていなかったときには、name 属性で指定されたインスタンスがフォーマットの対象となる。この場合は、そのインスタンス自身が java.math.BigDecimal 型であるか、あるいは java.lang.String 型(かつ右側の空白除去後に BigDecimal のコンストラクタによって解釈可能であるもの)のどちらかである必要がある。value 属性が指定されていた場合には、無視される。
property	-	name 属性で指定された bean においてアクセスされるプロパティの名前。value 属性が指定されていた場合には無視される。
scope	-	name 属性で指定された bean を検索する際のスコープ。
value	-	フォーマットする文字列。文字列は、右側の空白除去後に BigDecimal のコンストラクタによって解釈可能である必要がある。value 属性を指定した場合には、name 属性、および property 属性は無視される。
pattern	○	フォーマットする出力形式。pattern 属性で指定した出力形式は、DecimalFormat クラスのパターンとして解釈される。詳細は、DecimalFormat クラスのドキュメントを参照のこと。
scale	-	丸め動作後の小数点以下桁数。n を指定した場合には、小数第 n+1 位が丸められる。丸めモードは round 属性で指定する。round 属性が指定されていない場合は、四捨五入が行われる。
round	-	丸めモード。scale 属性が指定されている時、有効になる。ROUND_HALF_UP(四捨五入)、ROUND_FLOOR(切り捨て)、ROUND_CEILING(切り上げ)が設定可能である。デフォルトは ROUND_HALF_UP が実行される。これら3つの設定以外を指定した場合は、IllegalArgumentException がスローされる。

◆ 使用例

- Terasoluna Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC29 Decimal 表示」
 - ◇ /webapps/decimal/*
 - ◇ /webapps/WEB-INF/decimal/*
 - ◇ jp.terasoluna.thin.functionsample.decimal.*

■ WJ-08 トリム機能

◆ 概要

指定された文字列の半角スペース、または全角半角スペースを削除する。

◆ 解説

- `<t:rtrim>`
文字列の右側のスペースを削除する。
例： “ 文字列 ” ⇒削除⇒ “ 文字列”
- `<t:ltrim>`
文字列の左側のスペースを削除する。
例： “ 文字列 ” ⇒削除⇒ “文字列 ”
- `<t:trim>`
文字列の左右両側のスペースを削除する。
例： “ 文字列 ” ⇒削除⇒ “文字列”
- 全角スペースのトリム
`zenkaku` 属性に `true` を指定することで、全角スペースおよび半角スペースの両方を削除することができる。

◆ 使用方法例

```
<t:rtrim name="form0001" property="field001" zenkaku="true" />  
<t:ltrim name="form0001" property="field002" />  
<t:trim name="form0001" property="field003" />
```

◆ タグ属性一覧

- <t:rtrim>、<t:ltrim>、<t:trim>要素共通

属性	必須	概要
id	-	フォーマットした文字列を出力せずに、スクリプティング変数にセットする際に指定する。フォーマットされた文字列をスクリプティング変数にセットする場合には、filter 属性の指定に関わらず HTML 特殊文字はエスケープされない。
filter	-	フォーマットされた文字列を出力する際に、HTML 特殊文字をエスケープするかどうかを指定する。ただし、id 属性が指定されていた場合には、無視される。
ignore	-	name 属性で指定した bean が見つからなかったときに無視するかどうかを指定する。false を指定すると、bean が見つからなかったときに JspException が投げられる。
name	-	フォーマット対象の文字列をプロパティに持つ bean の名前。property 属性が指定されていなかったときには、name 属性で指定されたインスタンスの文字列表現 toString()メソッドで返される文字列)がフォーマットの対象となる。value 属性が指定されていた場合には、無視される。
property	-	name 属性で指定された bean においてアクセスされるプロパティの名前。value 属性が指定されていた場合には無視される。
scope	-	name 属性で指定された bean を検索する際のスコープ。
value	-	フォーマットする文字列。value 属性を指定した場合には、name 属性、および property 属性は無視される。
replaceSpToNbsp	-	この属性が true にセットされ、指定した bean プロパティの値に 1 Byte コードのスペースが存在する場合、 に置換する。無効にするためにはこの属性に明示的に false をセットする必要がある。ただし、id 属性が指定されていた場合には、無視される。
zenkaku	-	この属性が true にセットされた場合、半角スペースおよび全角スペースの削除を行う。

◆ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC30 トリム」
 - ◇ /webapps/trim/*
 - ◇ /webapps/WEB-INF/trim/*
 - ◇ jp.terasoluna.thin.functionsample.trim.*

■ WJ-09 文字列切り取り機能

◆ 概要

文字列の左端から指定された文字数分の文字列を切り出す。

◆ 解説

- <t:left>
String クラスの substring()メソッドによって、文字列の左端から指定された文字数を切り出す。

◆ 使用方法例

```
field001の内容  
「Java write once, Run anywhere.」
```

```
<t:left name="form0001"  
        property="field001"  
        length="10" />
```

↓上記の出力結果（左端から10文字が表示される。）

```
"Java write"
```

◆ タグ属性一覧

属性	必須	概要
id	-	フォーマットした文字列を出力せずに、スクリプティング変数にセットする際に指定する。フォーマットされた文字列をスクリプティング変数にセットする場合には、filter 属性の指定に関わらず HTML 特殊文字はエスケープされない。
filter	-	フォーマットされた文字列を出力する際に、HTML 特殊文字をエスケープするかどうかを指定する。ただし、id 属性が指定されていた場合には、無視される。
ignore	-	name 属性で指定した bean が見つからなかったときに無視するかどうかを指定する。false を指定すると、bean が見つからなかったときに JspException が投げられる。
name	-	フォーマット対象の文字列をプロパティを持つ bean の名前。property 属性が指定されていなかったときには、name 属性で指定されたインスタンスの文字列表現 toString()メソッドで返される文字列)がフォーマットの対象となる。value 属性が指定されていた場合には、無視される。
property	-	name 属性で指定された bean においてアクセスされるプロパティの名前。value 属性が指定されていた場合には無視される。
scope	-	name 属性で指定された bean を検索する際のスコープ。
value	-	フォーマットする文字列。value 属性を指定した場合には、name 属性、および property 属性は無視される。
replaceSpToNbsp	-	この属性が true にセットされ、指定した bean プロパティの値に 1 Byte コードのスペースが存在する場合、 に置換する。無効にするためにはこの属性に明示的に false をセットする必要がある。ただし、id 属性が指定されていた場合には、無視される。

◆ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC31 文字列切り取り」
 - ◇ /webapps/left/*
 - ◇ /webapps/WEB-INF/left/*
 - ◇ jp.terasoluna.thin.functionsample.left.*

■ WJ-10 コードリスト定義機能

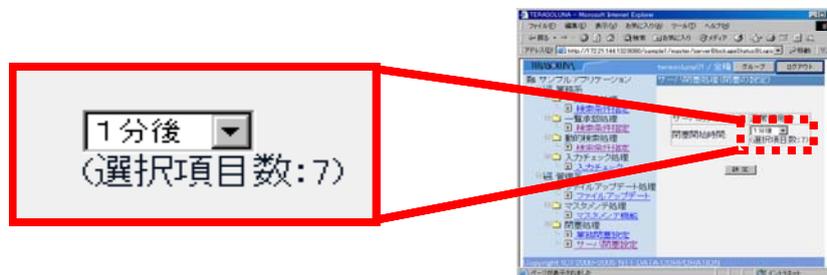
◆ 概要

読み込み済みのコードリストを、jsp 内で利用する。

コードリストの読み込みに関しては、『WB-05 コードリスト機能』を参照のこと。

◆ 解説

- `<t:defineCodeList>`
指定した id の CodeListLoader からコードリストを取得し、ページ属性の Bean として定義する。



◆ 使用方法例

```
<t:defineCodeList id="loader"/>
<html:select property="server_blockage_time">
  <html:options
    collection="loader" property="id" labelProperty="name" />
</html:select><br>
(選択項目数 : <t:defineCodeCount id="loader"/>)
```

◆ タグ属性一覧

属性	必須	概要
id	○	この属性からコードリストを検索する。このタグ宣言以降、<logic:iterator>タグ、<html:options>タグなどでコードリストが参照できる。

◆ 使用例

- Terasoluna Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC13 コードリスト」
 - ◇ /webapps/codelist/*
 - ◇ /webapps/WEB-INF/codelist/*
 - ◇ jp.terasoluna.thin.functionsample.codelist.*

■ WJ-11 コードリスト件数出力機能

◆ 概要

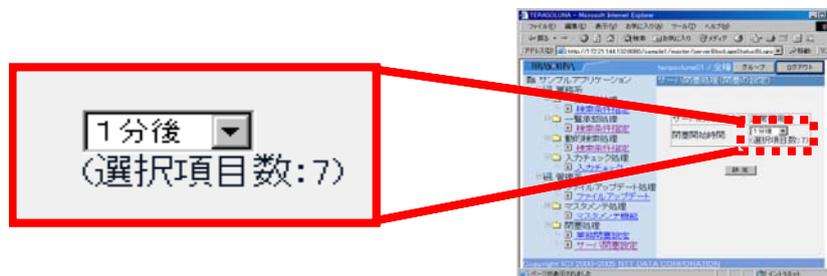
読み込み済みのコードリストの要素数を出力する。

コードリストの読み込みに関しては、『WB-05 コードリスト機能』を参照のこと。

◆ 解説

- <t:writeCodeCount>

指定した id の CodeListLoader からコードリストの要素数を出力する。



◆ 使用方法例

```
<t:defineCodeList id="loader"/>
<html:select property="server_blockage_time">
  <html:options
    collection="loader" property="id" labelProperty="name" />
</html:select><br>
(選択項目数: <t:writeCodeCount id="loader"/>)
```

◆ タグ属性一覧

属性	必須	概要
id	○	この属性で参照されるコードリストの要素数を出力する。コードリストが見つからない場合、0 が返却される。

◆ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC13 コードリスト」
 - ◇ /webapps/codelist/*
 - ◇ /webapps/WEB-INF/codelist/*
 - ◇ jp.terasoluna.thin.functionsample.codelist.*

■ WJ-12 指定コードリスト値表示機能

◆ 概要

読み込み済みのコードリストのコード値を指定し、画面に表示名を表示する。コードリストを前画面でセレクトボックスなどで表示し、ユーザが選択したコードを元に次画面で表示名を表示する場合に使用する。

コードリストの読み込みに関しては、『WB-05 コードリスト機能』を参照のこと。

◆ 解説

- `<t:writeCodeValue>`
CodeListLoader の id と表示したいコード値を指定することで画面にコードリストの表示名を表示する。



◆ 使用方法例

```
<tr>
  <th>客室タイプ</th>
  <td><t:writeCodeValue codeList="roomTypeCodeList"
    key="roomType01"/></td>
</tr>
```

◆ タグ属性一覧

属性	必須	概要
codeList	○	出力対象の CodeBean を保持している CodeListLoader インスタンスの BeanId。
key	-	取得したコードリストから値を取得するためのコード値を直接指定する。この値を省略した場合は必ず、name と property 属性を指定し、コード値を取得する Bean 名とプロパティ名を指定すること。
name	-	取得したコードリストから値を取得するためのコード値を保持する Bean の名前。key 属性が指定されていた場合は、無効。
property	-	取得したコードリストから値を取得するためのコード値を保持する Bean のプロパティ。key 属性が指定されていた場合は、無効。
scope	-	取得したコードリストから値を取得するためのコード値を保持する Bean が 存在するスコープ。

◆ 使用例

- Terasoluna Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC13 コードリスト」
 - ◇ /webapps/codelist/*
 - ◇ /webapps/WEB-INF/codelist/*
 - ◇ jp.terasoluna.thin.functionsample.codelist.*

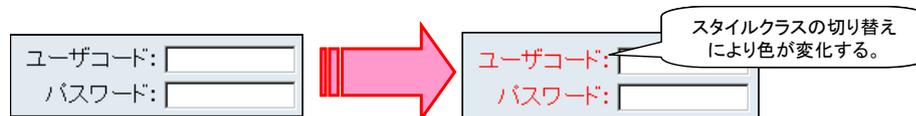
■ WK-01 スタイルクラス切り替え機能

◆ 概要

エラー時に、スタイルシートのクラスを切り替える。

◆ 解説

- `<ts:changeStyleClass>`
リクエストおよびセッションに指定されたフィールドに対するエラー情報が設定されているかどうかにより、スタイルシートのクラスを切り替える。



◆ 使用方法例

```
<td nowrap class='<ts:changeStyleClass name="userCode"
                    default="ItemLabel" error="ErrorItem"/>'>
  ユーザコード :
</td>
```

◆ タグ属性一覧

属性	必須	概要
name	○	エラー情報が設定されているかどうかを判定するフィールド名を指定する。
default	○	エラーがない場合のスタイルシートクラス名を指定する。
error	○	エラーがある場合のスタイルシートクラス名を指定する。

◆ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC33 スタイルクラス切り替え」
 - ◇ /webapps/styleclass/*
 - ◇ /webapps/WEB-INF/styleclass/*
 - ◇ jp.terasoluna.thin.functionsample.styleclass.*

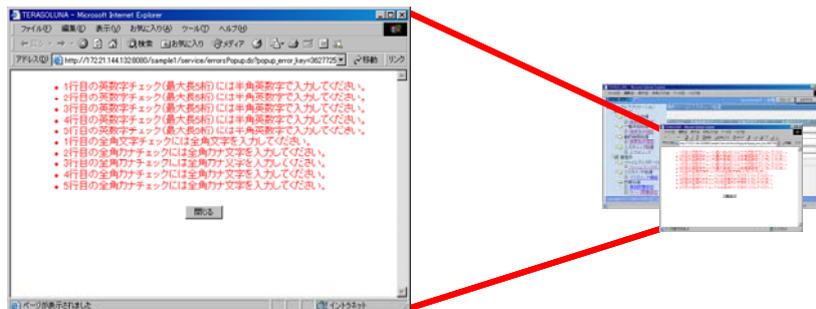
■ WK-02 メッセージ表示機能

◆ 概要

メッセージまたはエラーメッセージ情報を表示する。

◆ 解説

- <ts:errors>
Struts の ErrorsTag を拡張して、セッションからリクエストに移しかえられたエラー情報の取得・表示を行う。
- <ts:messages>
Struts の MessageTag を拡張して、セッションからリクエストに移しかえられた情報を取得する。



◆ 使用方法例

- <ts:errors>要素の使用例

```
<ts:errors/>
<html:button property="forward_action" value="閉じる"
onclick="window.close()"/>
```

- <ts:messages>要素の使用例

```
<ts:messages id="message" message="true">
  <bean:write name="message"/>
</ts:messages>
<html:button property="forward_actio
onclick="window.close()"/>
```

<ts:messages>要素は<ts:errors>要素と違い、メッセージを格納した Bean を定義するだけなので出力部分は実装しなければならない。

◆ タグ属性一覧

- <ts:errors>要素の属性一覧
<html:errors>要素の API と同様
- <ts:messages>要素の属性一覧
<html:messages>要素の API と同様
- <ts:errors>要素、<ts:messages>要素の注意点
ポップアップ画面での表示は、このタグが使用されない限り、セッションから情報は削除されない。

◆ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC34 メッセージ表示機能」
 - ◇ /webapps/message/*
 - ◇ /webapps/WEB-INF/message/*

■ WK-03 キャッシュ避け form タグ機能

◆ 概要

アクション URL にキャッシュ避け用ランダム ID を追加する。

◆ 解説

- `<ts:form>`
Struts の提供する `<html:form>` 要素を拡張する。
機能として、アクション URL にキャッシュ避け用ランダム ID を追加する。

◆ 使用方法例

```
<table border="0">
  <ts:form action="/logonBLogic">
    <div align="center">

↓出力結果

<table border="0">
  <form name="_logonForm" method="post"
        action="/sample1/logon/logonBLogic.do?r=3336517264997268823">
    <input type="hidden" name="org.apache.struts.taglib.html.TOKEN"
           value="6235eb8a1f477895315e96be95bcc7f2">
    <div align="center">
```

◆ タグ属性一覧

- `<html:form>` 要素の API と同様

◆ 使用例

- Terasoluna Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC35 拡張 form・リンク・submit」
 - ◇ /webapps/nocache/*
 - ◇ /webapps/WEB-INF/nocache/*
 - ◇ jp.terasoluna.thin.functionsample.nocache.*

- Terasoluna Server Framework for Java (Web 版) チュートリアル
 - 登録画面

■ WK-04 キャッシュ避けリンク機能

◆ 概要

Struts の提供する<html:link>タグを拡張する。

◆ 解説

- <ts:link>
アクション URL にキャッシュ避け用ランダム ID を追加する。

◆ 使用方法例

```
<ts:link href="/hoge.do">href=/hoge.do</ts:link>
```

↓出力時に、ランダムIDを追加する。

```
<a href="/hoge.do?r=3336517264997268823">href=/hoge.jsp</a>
```

◆ タグ属性一覧

- <html:link>要素の API と同様

◆ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC35 拡張 form ・ リンク ・ submit」
 - ◇ /webapps/nocache/*
 - ◇ /webapps/WEB-INF/nocache/*
 - ◇ jp.terasoluna.thin.functionsample.nocache.*
- TERASOLUNA Server Framework for Java (Web 版) チュートリアル
 - 「2.7 登録」
 - 登録画面

■ WK-05 フォームターゲット指定機能

◆ 概要

フォームのターゲットを指定する。

◆ 解説

- <ts:submit>
target 属性値を設定することで、フォームのターゲットを指定する。

◆ 使用方法例

```
<ts:submit value="submit" target="rightFrame"/>
```

◆ タグ属性一覧

属性	必須	概要
target	-	ターゲット先を指定する。

- その他の属性は<html:submit>と同様である。

◆ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC35 拡張 form・リンク・submit」
 - ◇ /webapps/nocache/*
 - ◇ /webapps/WEB-INF/nocache/*
 - ◇ jp.terasoluna.thin.functionsample.nocache.*

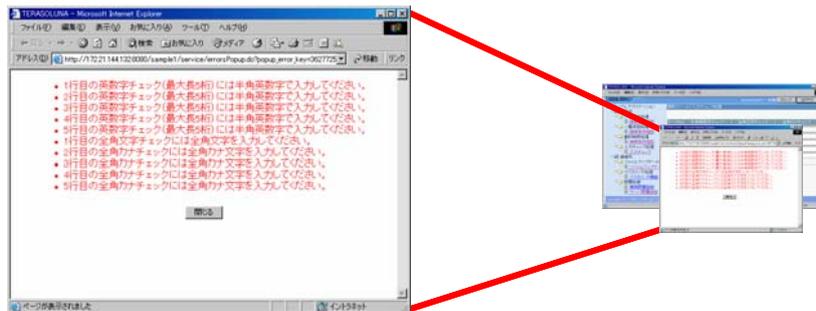
■ WK-06 メッセージポップアップ機能

◆ 概要

メッセージまたはエラーメッセージを表示するためのポップアップ画面を開く。

◆ 解説

- <ts:messagesPopup>
ポップアップ画面を開くためのスクリプトを、PageContext に設定する。
この機能を使用するためには、<ts:body>タグと連携させる必要がある。また、<ts:messagesPopup>タグは<ts:body>タグより前に記述する必要がある。
- <ts:body>
<ts:messagesPopup>タグによって設定されたスクリプトを画面のロード時に実行されるイベントに追加する。



◆ 使用方法例

```
<ts:messagesPopup popup="/service/errorsPopup.do" title="TERASOLUNA"/>
<ts:body>
↓上記の出力結果
<body onLoad="__onLoad__()">
  <script type="text/javascript">
  <!--
    function __onLoad__() {
      window.open("/sample1/service/errorsPopup.do","TERASOLUNA","");
    }
  //-->
</script>
```

◆ タグ属性一覧

● <ts:messagesPopup>

属性	必須	概要
popup	○	ポップアップ画面で表示する URL。JavaScript の window.open() の第一引数に対応する。
title	-	エラーを表示するポップアップ画面のタイトル。
param	-	JavaScript でポップアップ画面を開くときのパラメータ文字列。
paramType	-	JavaScript でポップアップ画面を開くときのパラメータ文字列を、ApplicationResources ファイルから取得する場合の リソースキー。
paramFunc	-	JavaScript でポップアップ画面を開くときのパラメータ文字列を取得する JavaScript 関数名。
windowId	-	開いたポップアップ画面を保持する JavaScript 変数名。

● <ts:body>

属性	必須	概要
onload	-	画面表示時に実行する JavaScript。
onunload	-	画面アンロード時に実行する JavaScript。
styleClass	-	スタイルシートのクラス名。
bgcolor	-	背景色。
background	-	背景に設定する画像
text	-	テキスト文字の色。
link	-	リンク部分の色。
vlink	-	既に選択されたリンク部分の色。
alink	-	選択中のリンク部分の色。

◆ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC32 メッセージポップアップ」
 - ◇ /webapps/popup/*
 - ◇ /webapps/WEB-INF/popup/*

■ WK-07 エラーメッセージチェック機能

◆ 概要

リクエスト、またはセッションにエラー情報が設定されているかどうかを判別して表示/非表示を切り替える。

◆ 解説

- `<ts:ifErrors>`
入力チェックエラーがある場合、あるいは出力パラメータにエラー情報が設定されている場合にタグのボディ部分を評価する。
- `<ts:ifNotErrors>`
入力チェックエラーがなく、かつ出力パラメータにエラー情報が設定されていない場合に、タグのボディ部分を評価する。

◆ 使用方法例

```
<ts:ifErrors>
    ... // エラーがある場合の表示項目等
</ts:ifErrors>
<ts:ifNotErrors>
    ... // エラーがない場合の表示項目等
</ts:ifNotErrors>
```

◆ タグ属性一覧

なし。

◆ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC34 メッセージ表示」
 - ◇ /webapps/message/*
 - ◇ /webapps/WEB-INF/message/*

■ WK-08 クライアントチェック拡張機能

◆ 概要

クライアントでの入力チェック時に検証に使用する値を javascript の変数として出力する。

入力チェックに関しては『WF-01 入力チェック拡張機能』を参照のこと。

◆ 解説

- <ts:javascript>
TERASOLUNA が提供する日本語系の検証ルールに必要な変数を javascript の変数として出力する

◆ 使用方法例

```
<meta http-equiv="Content-Type" content="text/html; charset=Windows-31J">
<title>入力チェック(クライアント)テスト画面</title>
<ts:javascript formName="/clientValidate"/>
</head>
<body>
```

◆ タグ属性一覧

<html:javascript>と同様

◆ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC36 クライアントチェック」
 - ◇ /webapps/clientvalidation/*
 - ◇ /webapps/WEB-INF/clientvalidation/*

■ WK-09 一覧表示関連機能

◆ 概要

一覧表示は、Struts が提供する一覧表示機能<logic:iterate>要素を使用する。表示された一覧表に対して、ページリンク機能<ts:pageLinks>要素を使用することができる。一覧表示機能、ページリンク機能ともに一画面に対して複数記述することができる。

◆ 解説

- <logic:iterate>
Struts の一覧表示機能<logic:iterate>要素を使用する。属性にて指定された Bean から一覧情報を取得して、取得した一覧情報分のループを回す。指定する Bean は、Collection、ArrayList、Vector、Enumeration、Iterator、Map、HashMap、Hashtable、TreeMap、配列などである。繰り返し項目（HTML テーブル内の<TR>～</TR>など）を囲むように記述する。詳細な使用方法は Struts を参照のこと。
- <ts:pageLinks>
<logic:iterate>要素によって定義された一覧のページ遷移のリンクを表示する。ページリンク機能を使用する場合は、アクションフォームに以下のプロパティを用意する必要がある。
 - 表示行数を保持するプロパティ
 - 表示開始インデックスを保持するプロパティ
 - 一覧情報全行数を保持するプロパティ
- 一覧表示関連機能の詳細な使用方法、タグ属性などに関しては、「WI-01 一覧表示機能」を参照のこと。

◆ 使用例

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC24 一覧表示」
 - ◇ /webapps/pagelink/*
 - ◇ /webapps/WEB-INF/pagelink/*
 - ◇ jp.terasoluna.thin.functionsample.pagelink.*
- TERASOLUNA Server Framework for Java (Web 版) チュートリアル
 - 「2.5 一覧表示」
 - 一覧表示画面