



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

機能説明書

第 2.0.5.0 版

NTT Data

株式会社 NTTデータ

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

- (1)本ドキュメントの著作権及びその他一切の権利は、NTT データあるいは NTT データに権利を許諾する第三者に帰属します。
- (2)本ドキュメントの一部または全部を、自らが使用する目的において、複製、翻訳、翻案することができます。ただし本ページの規約全文、および NTT データの著作権表示を削除することはできません。
- (3)本ドキュメントの一部または全部を、自らが使用する目的において改変したり、本ドキュメントを用いた二次的著作物を作成することができます。ただし、「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確性や信頼性の保証、及び瑕疵担保義務も含め、直接、間接に被ったいかなる損害に対しても一切の責任を負いません。
- (7)NTT データは、本ドキュメントが第三者の著作権、その他如何なる権利も侵害しないことを保証しません。また、著作権、その他の権利侵害を直接又は間接の原因としてなされる如何なる請求(第三者との間の紛争を理由になされる請求を含む。)に関しても、NTT データは一切の責任を負いません。

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

- Terasoluna は、株式会社 NTT データの登録商標です。
- その他の会社名、製品名は、各社の登録商標または商標です。

本ドキュメントは、表紙に示す版の TERASOLUNA Server Framework for Java (Web 版) に対応しています。

変更履歴

バージョン	日付	改訂箇所	改訂内容
2.0.2.0	2009/03/31	商標、説明	富士通 Interstage に対応
〃	〃	CE-01 メッセージ管理機能 WG-01 メッセージ管理機能	メッセージ管理機能に関する記述を修正
〃	〃	WC-01 例外ハンドリング機能	SystemExceptionHandler に logLevel 属性を追加 DefaultExceptionHandler を追加
〃	〃	WJ-05 日付変換機能 WJ-06 和暦日付変換機能	value 属性の説明を修正 format 属性を追加
〃	〃	CB-01 データベースアクセス機能	QueryRowHandleDAO に関する記述を追加
〃	2009/05/01	WJ-01 アクセス権限チェック機能	<t:ifAuthorizedBlock>タグの解説を修正 <t:ifAuthorized>タグ、<t:ifAuthorizedBlock>タグ の使用方法例を修正
〃	2009/09/30	CA-01 トランザクション管理機能	JTA に関する記述を変更
〃	〃	CD-01 ユーティリティ機能	PropertyUtil の説明を修正
〃	〃	WB-05 コードリスト機能	MappedCodeListLoader に関する記述を追加
2.0.2.1	2010/02/26	CE-01 メッセージ管理機能	トランザクションインタセプタのロールバック対象 例外の設定で Throwable が不要であることを追記
〃	〃	全体	Bean 定義例の書式を統一
〃	〃	CA-01 トランザクション管理機能 CE-01 メッセージ管理機能	BeanID の表記を統一
〃	〃	CD-01 ユーティリティ機能	和暦のプロパティ設定について説明を追記
〃	〃	CE-01 メッセージ管理機能	xml 定義の ref 参照部分を system-messages, application-messages に変更
〃	〃	CD-01 ユーティリティ機能	誤字を修正
〃	〃	WB-04 フォームプロパティリセット機能	チェックボックス、ラジオボタン以外のフィールド 初期化について説明を追記
〃	〃	WK-02 メッセージ表示機能	<ts:errors><ts:messages>タグの使用方法を追記
〃	〃	WJ-01～WK-09 画面表示機能	<ts:submit>タグの target 属性の説明を追記
〃	〃	全体	フッター情報を変更
2.0.3.0	2010/04/02	CA-01 トランザクション管理機能	ポイントカット指定に関する既知の不具合について Spring2.5.6 により解消のため修正
〃	〃	CB-01 データベースアクセス機能	バッチ処理の Oracle 実装の更新件数について、参 考資料を 11g 用に URL を修正
2.0.3.1	2011/07/04	WA-06 セッション同期化機能	新規追加

変更履歴

バージョン	日付	改訂箇所	改訂内容
2.0.4.0	2012/04/02	WI-01 一覧表示機能	総件数の出力例を修正
〃	〃	AL008-01 リクエストパラメータトリム機能 AL008-02 エンコードフィルタ機能 AL009-01 デバッグログ出力機能 AL009-02 操作ログ出力機能 AL009-03 DB アクセスログ出力機能 AL010 共通例外機能 AL017 ビジネスロジック入出力定義設定ファイル削減機能 AL018 設定ファイルワイルドカード指定読み込み機能 AL020 RequestProcessor 拡張性向上機能 AL046 共通画面フロー機能 AL047 動的コードリスト値抽出・表示機能 AL055 拡張ページリンク機能	新規追加
2.0.5.0	2012/12/26	全体	フッター情報を変更
〃	〃	WF-01 拡張入力チェック機能	数値チェックについて注意事項を追記
〃	〃	WG-01 メッセージ管理機能	誤字を修正

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
CE-01	メッセージ管理機能

● 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
WA-06	セッション同期化機能
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	例外ハンドリング機能
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	メッセージ管理機能
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

AL008-01	リクエストパラメータトリム機能
AL008-02	エンコードフィルタ機能
AL009-01	デバッグログ出力機能
AL009-02	操作ログ出力機能
AL009-03	DB アクセスログ出力機能
AL010	共通例外機能
AL017	ビジネスロジック入出力定義設定ファイル削減機能
AL018	設定ファイルワイルドカード指定読み込み機能
AL020	RequestProcessor 拡張性向上機能
AL046	共通画面フロー機能
AL047	動的コードリスト値抽出・表示機能
AL055	拡張ページリンク機能

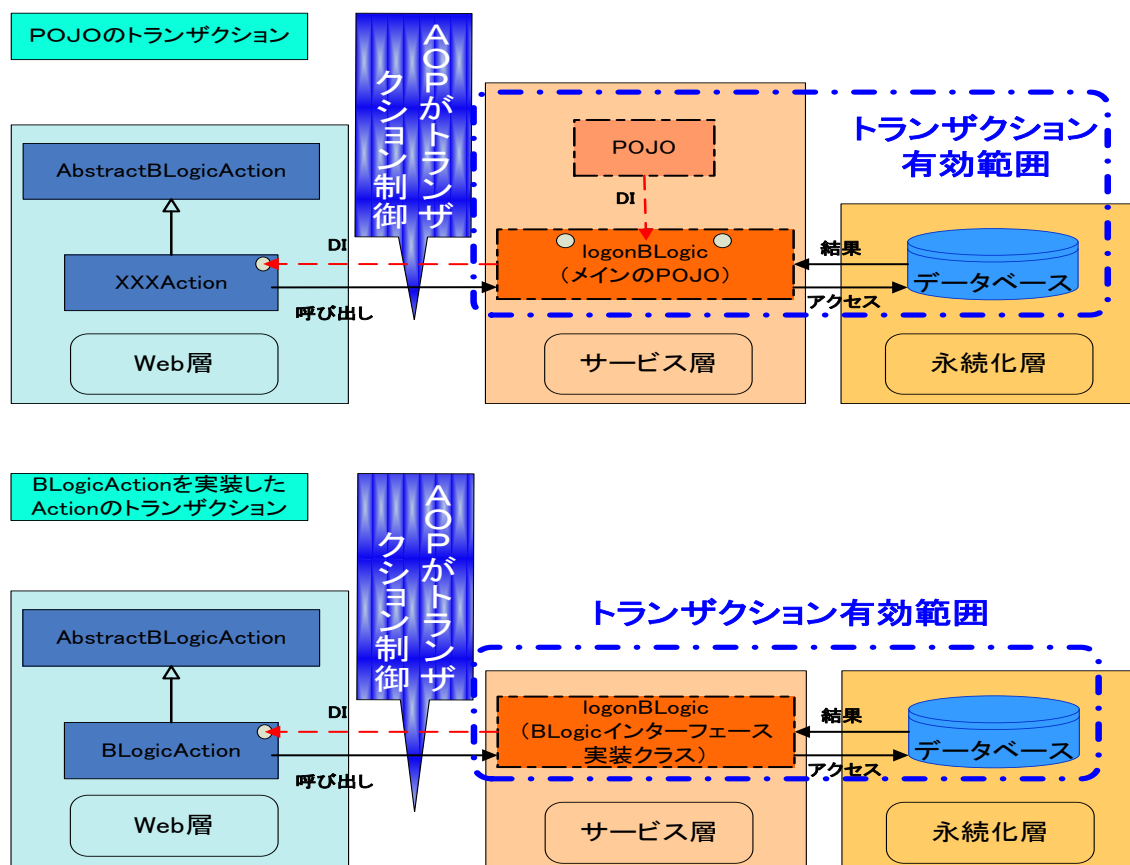
CA-01 トランザクション管理機能

■ 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。デフォルトの設定では、検査例外の場合はコミットされ、非検査例外、エラーの場合はロールバックされる。

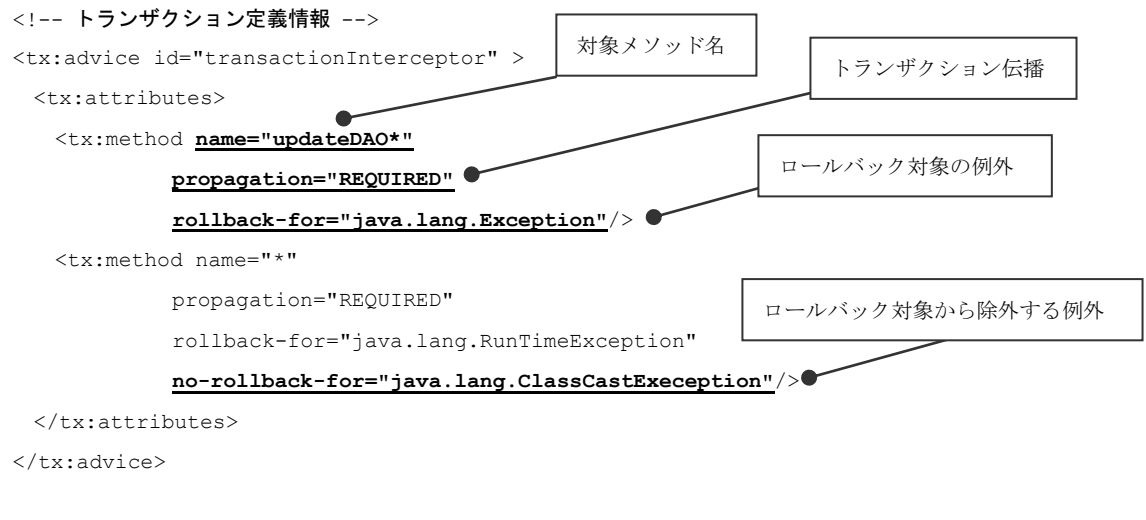
例外をスローせずにロールバックを行いたい場合は後述の TransactionUtil クラスの setRollbackOnly メソッドが利用できる。

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。なお、Spring の宣言的トランザクション管理機能では `Error` や `RuntimeException` が発生するとデフォルトでロールバックするようになっているため、`rollback-for` に `java.lang.Throwable` に設定する必要はない。

- 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 BeanID 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)

<!-- DataSourceの設定。 -->

```
<bean id="dataSource" class=".....">.....</bean>
```

<!-- 単一のJDBC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.jdbc.datasource.DataSourceTransactionManager">  
  <property name="dataSource" ref="dataSource" />  
</bean>
```

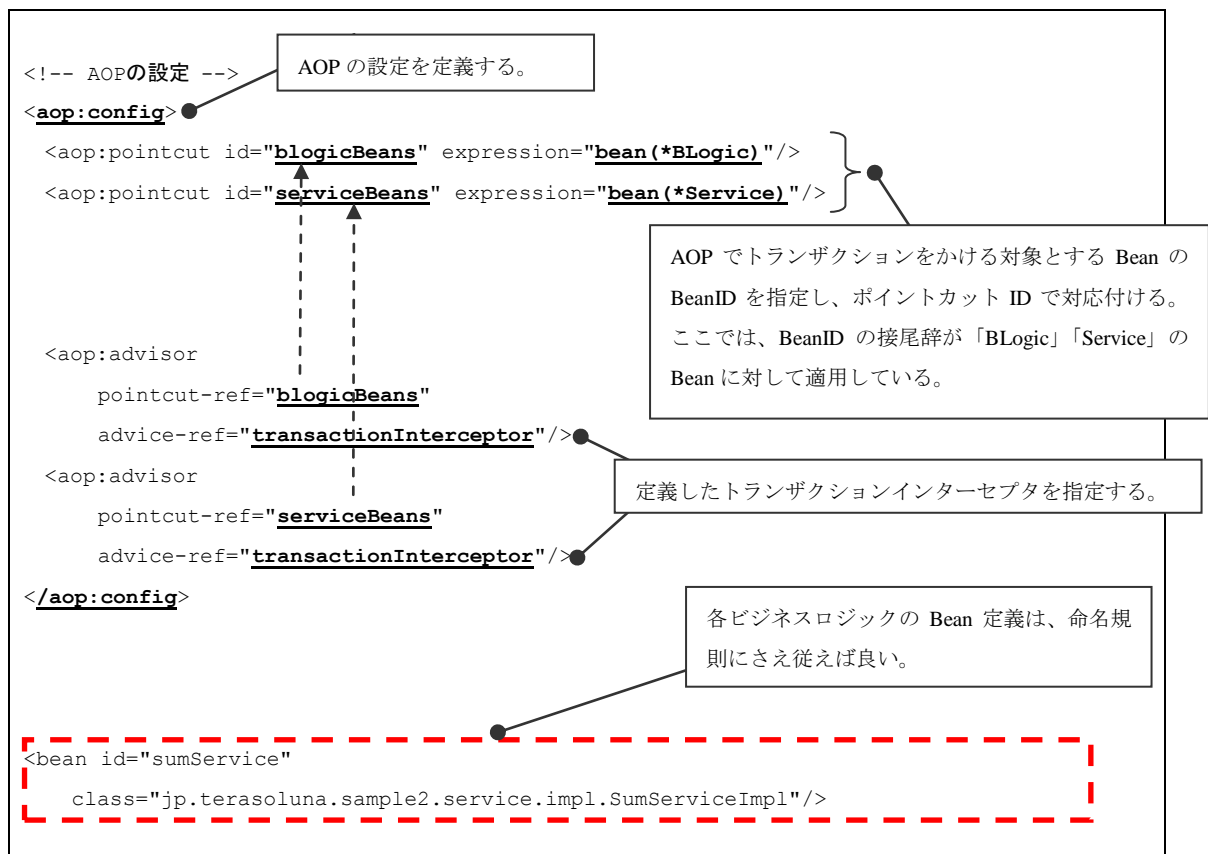
トランザクションインターセプタを定義する。

```
<tx:advice id="transactionInterceptor" transaction-manager="transactionManager">  
  <tx:attributes>  
    <tx:method name="insert*"  
      propagation="REQUIRED"  
      rollback-for="java.lang.Exception"/>  
    <tx:method name="*"   
      propagation="REQUIRED"  
      read-only="true"/>  
  </tx:attributes>  
</tx:advice>
```

トランザクションマネージャを指定する。
省略時 “transactionManager”

トランザクション定義情報の設定。

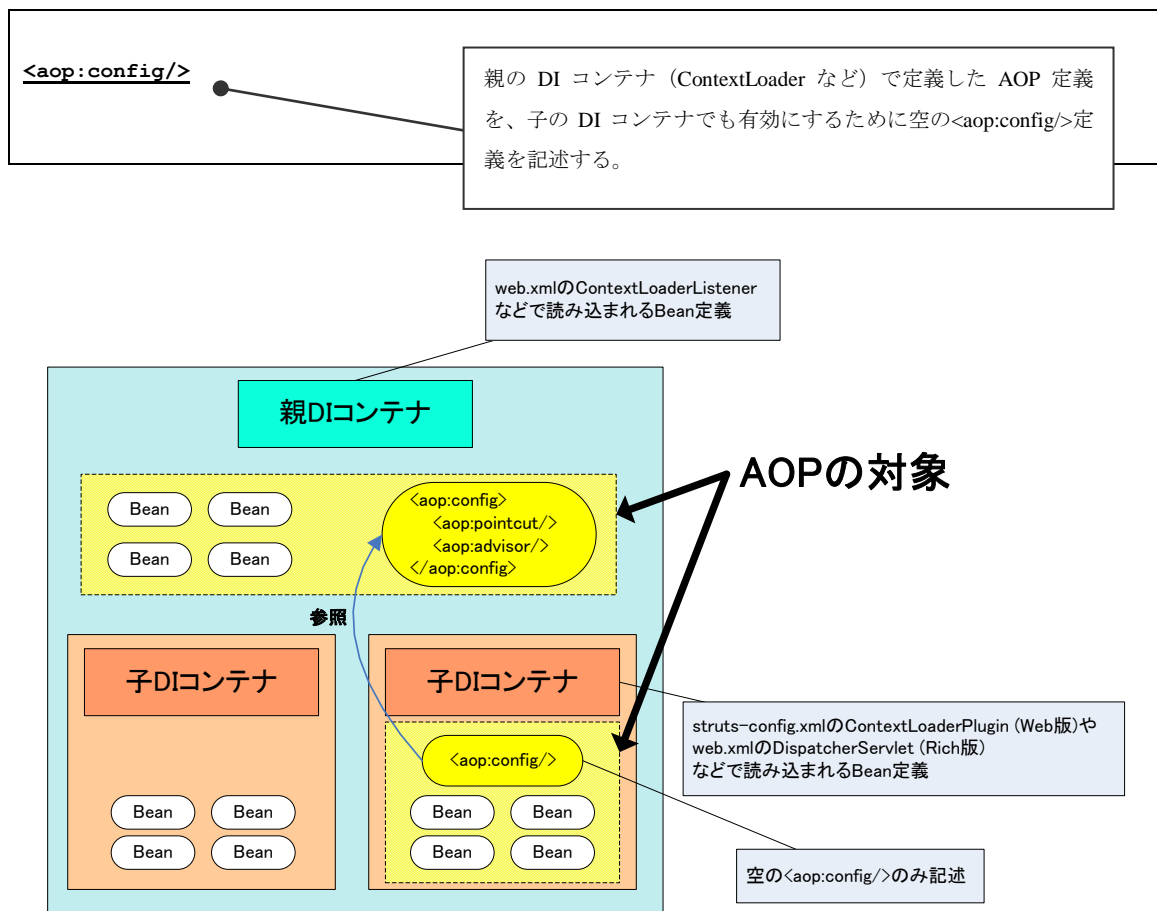
(続く)



- 親の DI コンテナ（ContextLoader で読み込まれる Bean 定義ファイルなど）で定義されたアドバイス定義は、子の DI コンテナでも有効である。ただし、親で定義されたアドバイスを子の DI コンテナで有効にするために、空の `<aop:config/>` 定義が必要となる。

親の DI コンテナで定義したアドバイス定義を子の DI コンテナでも同様に定義した場合、アドバイスが二重にウィービングされるので注意が必要である。

（トランザクションインタセプタが多重にウィービングされると、後述の `setRollbackOnly` メソッドで問題が発生する）



- 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**。(TERASOLUNA のデフォルト設定の場合、**検査例外もロールバックされる**)

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。

※後述の TransactionUtil#setRollbackOnly を利用する際の注意事項も参照のこと

- 複数 DB 使用時

JTA をサポートするアプリケーションサーバを利用して、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。また、各アプリケーションサーバの JTA サポート状況は各ベンダにお問い合わせ下さい。

➤ 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="queryDAO" />
</bean>
```

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

● (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

- **TransactionUtil#setRollbackOnly を利用する際の注意事項**

1. 本ユーティリティを利用する際は AOP を利用して TransactionInterceptor が対象の処理にウィービングされている必要がある。
2. トランザクション対象の処理が多段階にネストするような場合、下位の処理で setRollbackOnly を利用した場合、上位の処理でも setRollbackOnly を呼ぶ必要がある。上位の処理で setRollbackOnly を呼ばない場合は、ロールバックは行われるが例外が発生する。設定ミス等で単一の処理に対して多重に TransactionInterceptor がウィービングされた場合も例外が発生する。
3. 本ユーティリティを利用した処理を JUnit で単体試験を行う場合は注意が必要である。通常通り作成したテストケースのままでは、NoTransactionException が発生してしまう。これは TransactionStatus が参照できない為である。TransactionStatus は通常 TransactionInterceptor によって生成される。

正常にテストを行うには、擬似的に TransactionStatus を生成する必要がある。以下のようなモッククラスを使うことで、擬似的に TransactionStatus を生成し、テストを行うことができる。

```
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.interceptor.DefaultTransactionAttribute;
import org.springframework.transaction.interceptor.TransactionAspectSupport;
import org.springframework.transaction.interceptor.TransactionAttribute;
import org.springframework.transaction.support.SimpleTransactionStatus;

public class MockTransactionAspectSupport extends TransactionAspectSupport {
    private TransactionStatus status = null;

    public MockTransactionAspectSupport() {
        TransactionAttribute txAttr = new DefaultTransactionAttribute();
        status = new SimpleTransactionStatus();
        String joinpointIdentification = null;
        this.prepareTransactionInfo(txAttr, joinpointIdentification, status);
    }

    public boolean isRollbackOnly() {
        return status.isRollbackOnly();
    }
}
```

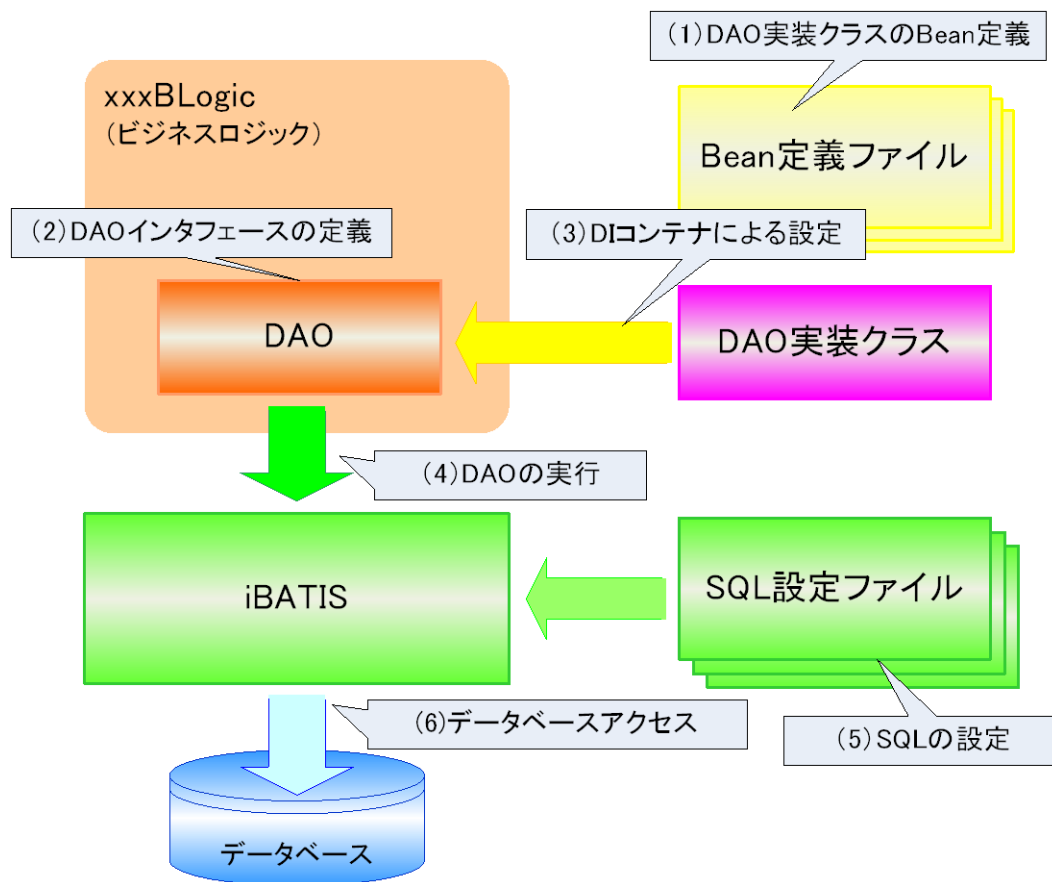

CB-01 データベースアクセス機能

■ 概要

◆ 機能概要

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

◆ 概念図



◆ 解説

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

■ 使用方法

◆ コーディングポイント

- データソースの Bean 定義

データソースの設定はアプリケーション Bean 定義ファイルに定義する。

- アプリケーション Bean 定義ファイル

- ✧ JNDI の場合 JndiObjectFactoryBean を使用する。

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

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

◇ JNDI 名が頻繁に変わる場合

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

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

Property-placeholder を定義する。

<!-- JNDI 関連のプロパティ -->

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

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

context スキーマを定義する。

プロパティファイルの例 (jndi.properties)

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

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

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

Property-placeholder を定義する。

<!-- JDBC関連のプロパティ -->

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

スキーマを定義する。

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

クラスパスを指定する。

DriverManagerDataSource
の定義。

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

➤ プロパティファイルの例 (jdbc.properties)

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

- iBATIS マッピング定義ファイル

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

- Select 文の実行例

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

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

- Insert、Update、Delete 文の実行

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

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

- Procedure 文の実行

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

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

- iBATIS 設定ファイル

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

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

- SqlMapClientFactoryBean の Bean 定義

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

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

- iBATIS 設定ファイル

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

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

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

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


- DAO の Bean 定義

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

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

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

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

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

- QueryDAOiBatisImpl のメソッドの戻り値の指定

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

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

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

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

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

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

- QueryDAOiBatisImpl を使用した一覧データ取得例

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

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

- Bean 定義ファイル

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

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

➤ Bean 定義ファイル

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

➤ ビジネスロジック

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

一覧表示の詳細な使用方法は、『WI-01 一覧表示機能』を参照のこと。

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

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

設定された DAO を使用して、データベースから一覧情報を取得する。

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

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

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

設定された DAO を使用して、データベースから一覧情報を取得する。

- UpdateDAOiBatisImpl を使用したデータ登録例

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

- ビジネスロジック

Bean 定義ファイルにて設定された DAO のメソッドを使用する。

```
private UpdateDAO updateDAO = null;

.....Setterは省略

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

設定された DAO を使用して、データベースにデータを登録する。

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

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

詳細は UpdateDAOiBatisImpl の JavaDoc を参照のこと。

- ビジネスロジック

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

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

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

- 注意点

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

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

「標準バッチ処理の Oracle 実装の更新件数」を参照のこと。

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

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

- ビジネスロジック

Bean 定義ファイルにて設定された DAO のメソッドを使用する。

```
private StoredProcedureDAO spDAO = null;

.....Setterは省略

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

設定された DAO を使用してプロシージャを実行する。

- iBATIS マッピング定義ファイル

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

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

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

- 実行するプロシージャ（Oracle を利用した例）

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

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

➤ DataRowHandler の実装

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

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

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

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

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

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

```
        }
```

```
    }
```

```
}
```

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

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

➤ ビジネスロジック

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

```
.....Setterは省略
```

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

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

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

```
    queryRowHandleDAO.executeWithRowHandler(
```

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

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

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

```
    return result;
```

```
}
```

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

◆ 拡張ポイント

なし。

■ 構成クラス

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

■ 関連機能

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

■ 使用例

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

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

- 「UC01 データベースアクセス」

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

■ 備考

- 大量データを検索する際の注意事項

iBATIS マッピング定義ファイルの<statement>要素、<select>要素、<procedure>要素にて大量データを返すようなクエリを記述する場合には、fetchSize 属性に適切な値を設定しておくこと。

fetchSize 属性には JDBC ドライバとデータベース間の通信において、一度の通信で取得するデータの件数を設定する。fetchSize 属性を省略した場合は各 JDBC ドライバのデフォルト値が利用される。

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

CC-01 JNDI アクセス機能

■ 概要

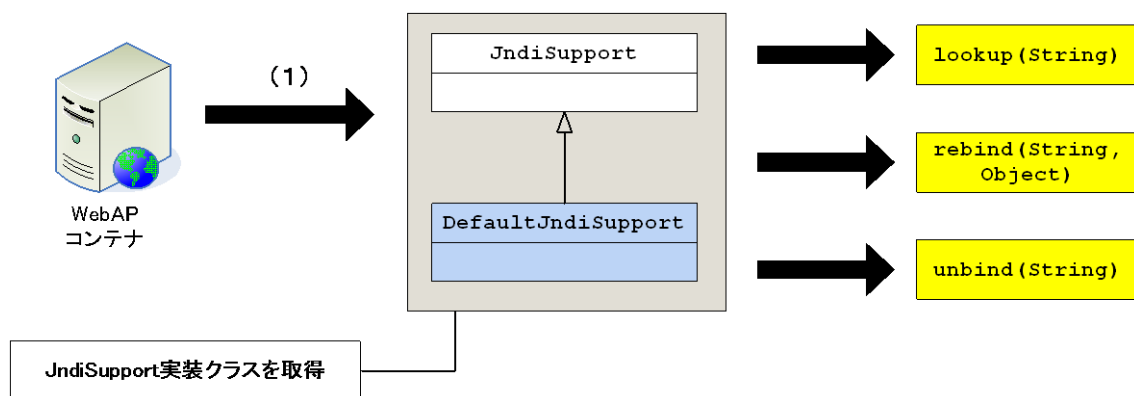
◆ 機能概要

- 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。

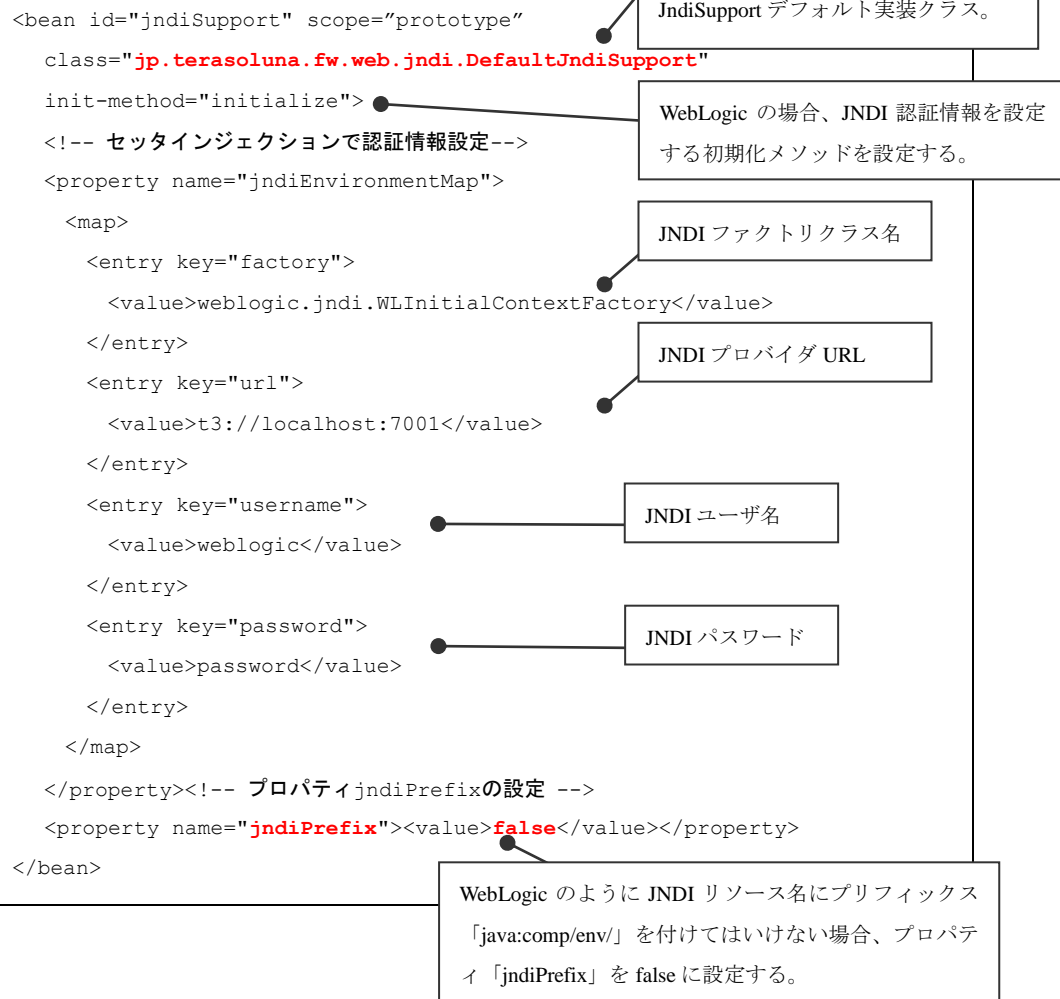
■ 使用方法

◆ 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="jndiSupport" />
</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
getSystemTime()	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` または `ApplicationResources.properties` の記述より追加で読み込まれたプロパティファイルに元号名、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。(デフォルトでは、元号名等は既に `system.properties` ファイルに記述されている)

● `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。

※`rmdirs(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の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>rmdirs(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指定する。

● 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。
getPropertiesValues(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。
getPropertiesValues(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
```

- ApplicationResources.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。  
// ファイル名には.propertiesが付いていても付いてなくてもよい。  
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("error.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。  
// ファイル名には.propertiesが付いていても付いてなくてもよい。  
// (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、test.message.01のプロパティ値、  
// error.message.03のプロパティ値のセットを取得する。  
// (結果: set2には"テストメッセージ1"、"エラーメッセージ3"が格納される)  
Enumeration en4 = new StringTokenizer("test.message.01 error.message.03");  
Set set2 = PropertyUtil.getPropertiesValues(prop, en4);
```

■ 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 = parseCSV("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");
```

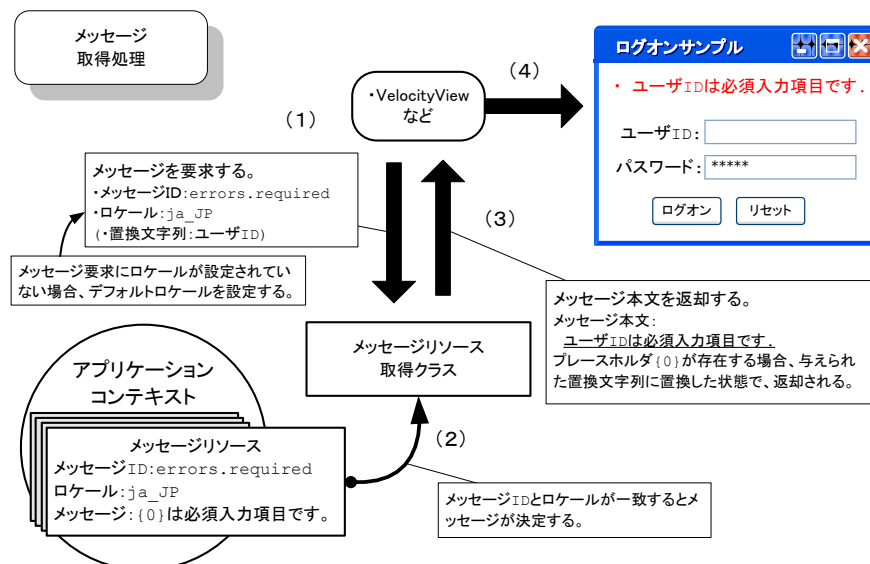
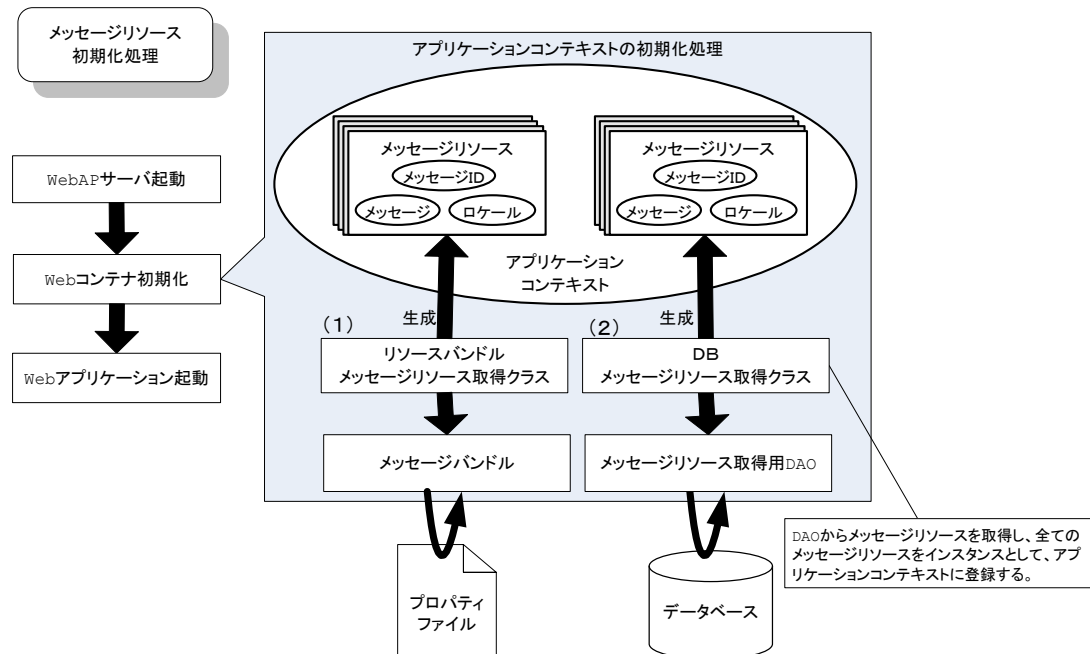
CE-01 メッセージ管理機能

■ 概要

◆ 機能概要

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

◆ 概念図



◆ 解説

- メッセージリソース初期化处理

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

- メッセージリソース取得処理

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

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

■ 使用方法

◆ コーディングポイント

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

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

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

messageSource を指定する。

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

プロパティファイルはクラスパス上に配置する。

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

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

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

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

➤ メッセージリソース取得 Bean の定義

TERASOLUNA Server Framework for Java が提供している

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

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

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

messageSource を指定する。

DataSourceMessageSource を指定する。

利用するメッセージ取得用の DAO を指定する。

➤ メッセージリソース取得用 DAO の Bean 定義

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

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

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

```
<bean id="dbMessageResourceDAO"
class="jp.terasoluna.fw.message.DBMessageResourceDAOImpl">
  <property name="dataSource" ref bean="dataSource"/>
</bean>
<bean id="dataSource" .....>
</bean>
```

DBMessageResourceDAOImpl を指定する。

dataSource を指定する。

➤ データソースの定義

『CB-01 データベースアクセス機能』を参照して設定する。

➤ メッセージ文字列の定義

メッセージ文字列はデータベース中の以下のテーブルに格納しておく：

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

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

```
SELECT CODE,MESSAGE FROM MESSAGES
```

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

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

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

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

org.springframework.context.support ApplicationObjectSupport

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

- メッセージ使用例

- メッセージ取得クラスインタフェースサンプル

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

業務共通機能担当者が作成する。

ビジネスロジック開発者が使用するメソッドを規定する。

- メッセージ取得クラス実装クラスサンプル

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

業務共通機能担当者が作成する。

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

➤ ビジネスロジックサンプル

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

```
    //メッセージ出力クラス用setter
```

```
    MessageAccessor msgAcc = null;
```

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

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

```
    }
```

```
    //ビジネスロジック
```

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

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

```
        String outPutMessage = null;
```

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

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

```
        return result;
```

```
    }
```

```
}
```

ビジネスロジック開発担当者が作成する。

上記、メッセージ出力クラスを DI するためのSetterを記述する。

メッセージ出力クラスからメッセージ取得メソッドを利用する。

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

```
<!--メッセージ出力クラス -->
```

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

```
<!-- 業務ロジッククラス -->
```

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

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

```
</bean>
```

業務共通機能担当者が記述する。

ビジネスロジック開発担当者が記述する。

➤ メッセージリソースの再定義方法

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

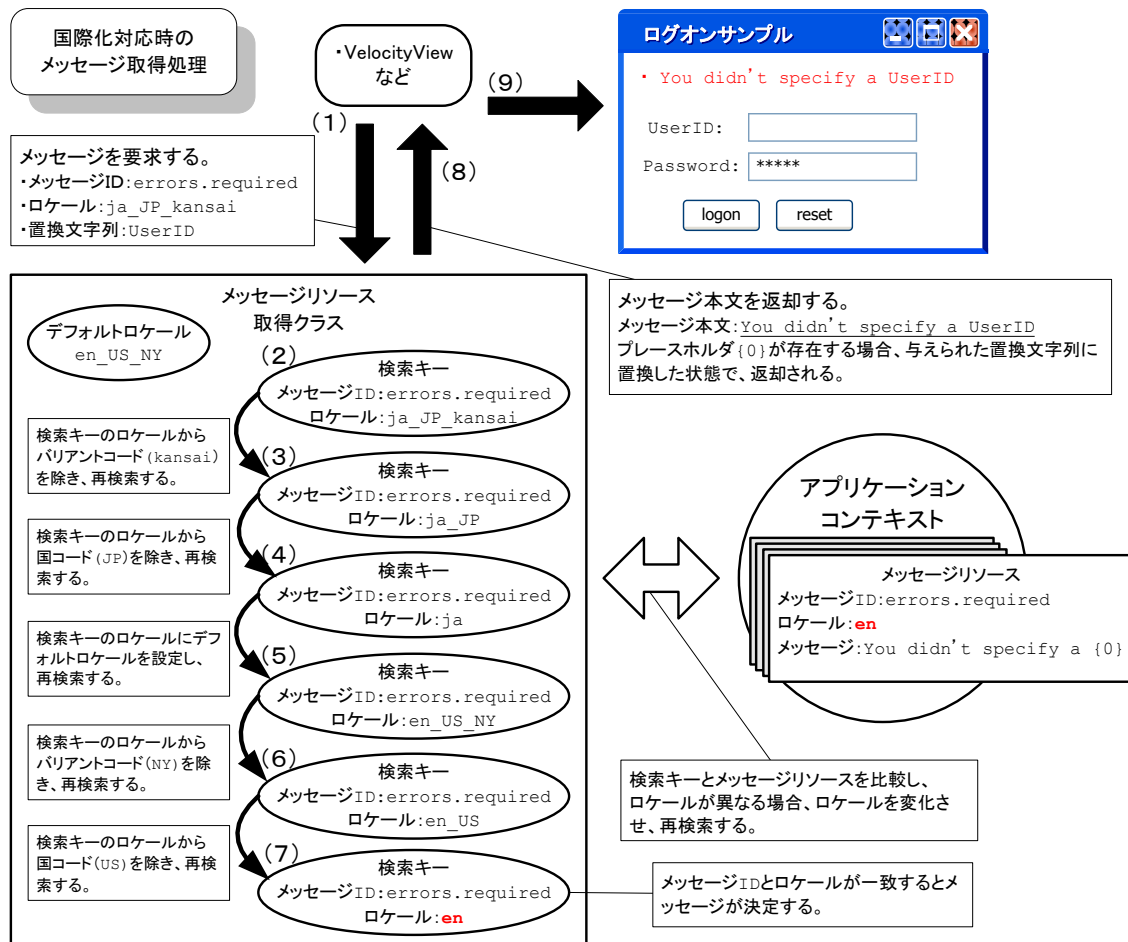
**jp.terasoluna.fw.message.DataSourceMessageSource クラスの
reloadDataSourceMessage メソッド**

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

■ メッセージの国際化対応

◆ 国際化対応時のメッセージ決定ロジック

● 概要図



● 解説

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

バリエーションコードを除き、再検索する。

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

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

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

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

デフォルトロケールを指定する。

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

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

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

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

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

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

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

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

メッセージ取得 SQL 文のフォーマットは以下の通り。

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

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

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

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

国際化対応する場合のみ設定。
言語コードのカラム名を指定する。

国際化対応する場合のみ設定。
国コードのカラム名を指定する。

国際化対応する場合のみ設定。
バリエントコードのカラム名を指定する。

DBのテーブル名及びカラム名は以下の様な設定となる。

テーブル名 = DBMESSAGES

メッセージコードを格納するカラム名 = BANGOU

メッセージの言語コードを格納するカラム名 = GENGO

メッセージの国コードを格納するカラム名 = KUNI

メッセージのバリエントコードを格納するカラム名 = HOUGEN

メッセージ本文を格納するカラム名 = HONBUN

検索SQL文は以下の通り。

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

■ リファレンス

◆ 構成クラス

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

◆ 拡張ポイント

- なし

■ 関連機能

- なし

■ 使用例

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

■ 備考

◆ メッセージリソースのテーブルスキーマをデフォルト値から変更する

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

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

メッセージ取得 SQL 文の SELECT 節のフォーマットは以下の通り。

SELECT メッセージコード, メッセージ本体 **FROM** テーブル名

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

例) データベースのテーブル名及びカラム名を以下の様にする場合

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

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

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

テーブル名を指定する。

メッセージコードのカラム名を指定する。

メッセージ本文のカラム名を指定する。

メッセージ取得 SQL 文は以下の通り。

SELECT BANGOU,HONBUN **FROM** DBMESSAGES

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

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

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

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

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

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

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

テーブル名を指定する。

メッセージコードのカラム名を指定する。

メッセージ本文のカラム名を指定する。

検索 SQL 文を指定する。

◆ 第2メッセージリソースの使用

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

☆ 第2メッセージリソース取得 Bean の定義

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

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

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

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

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

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

次に参照される

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

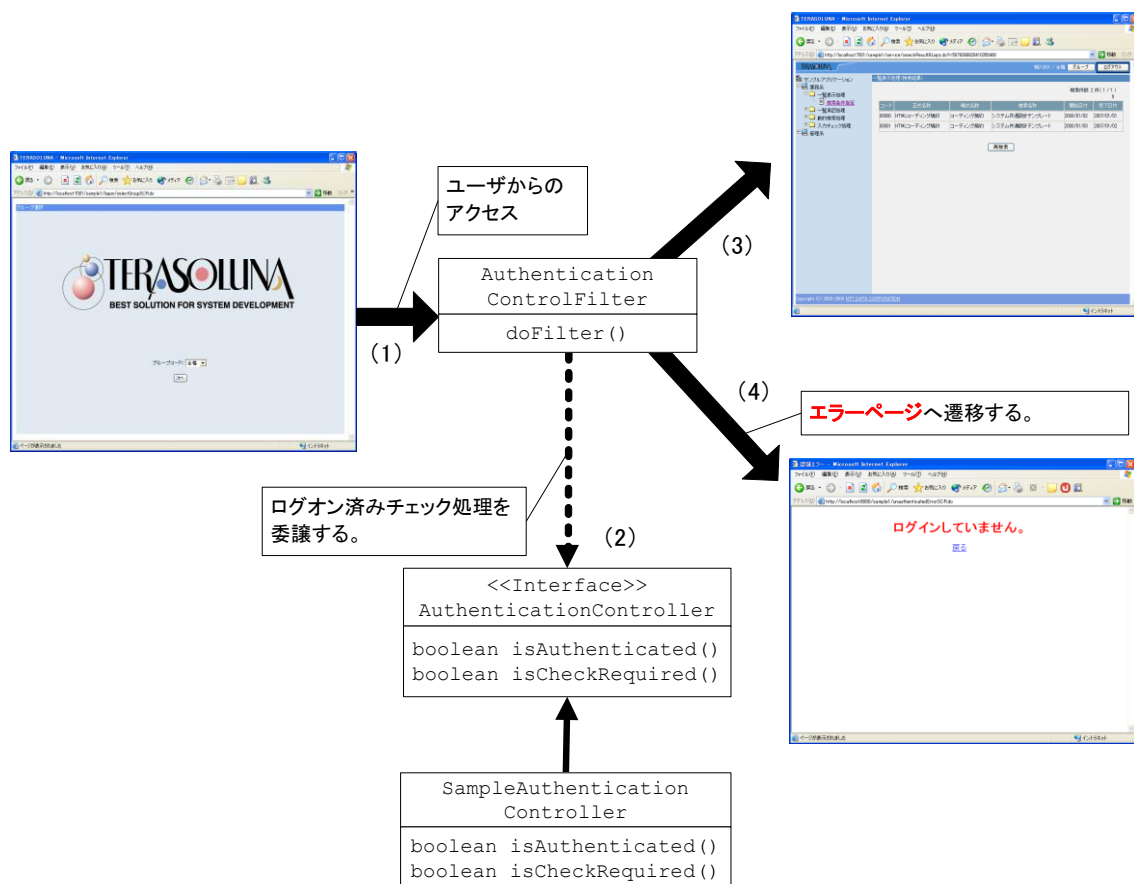
WA-01 ログオン済みチェック機能

■ 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

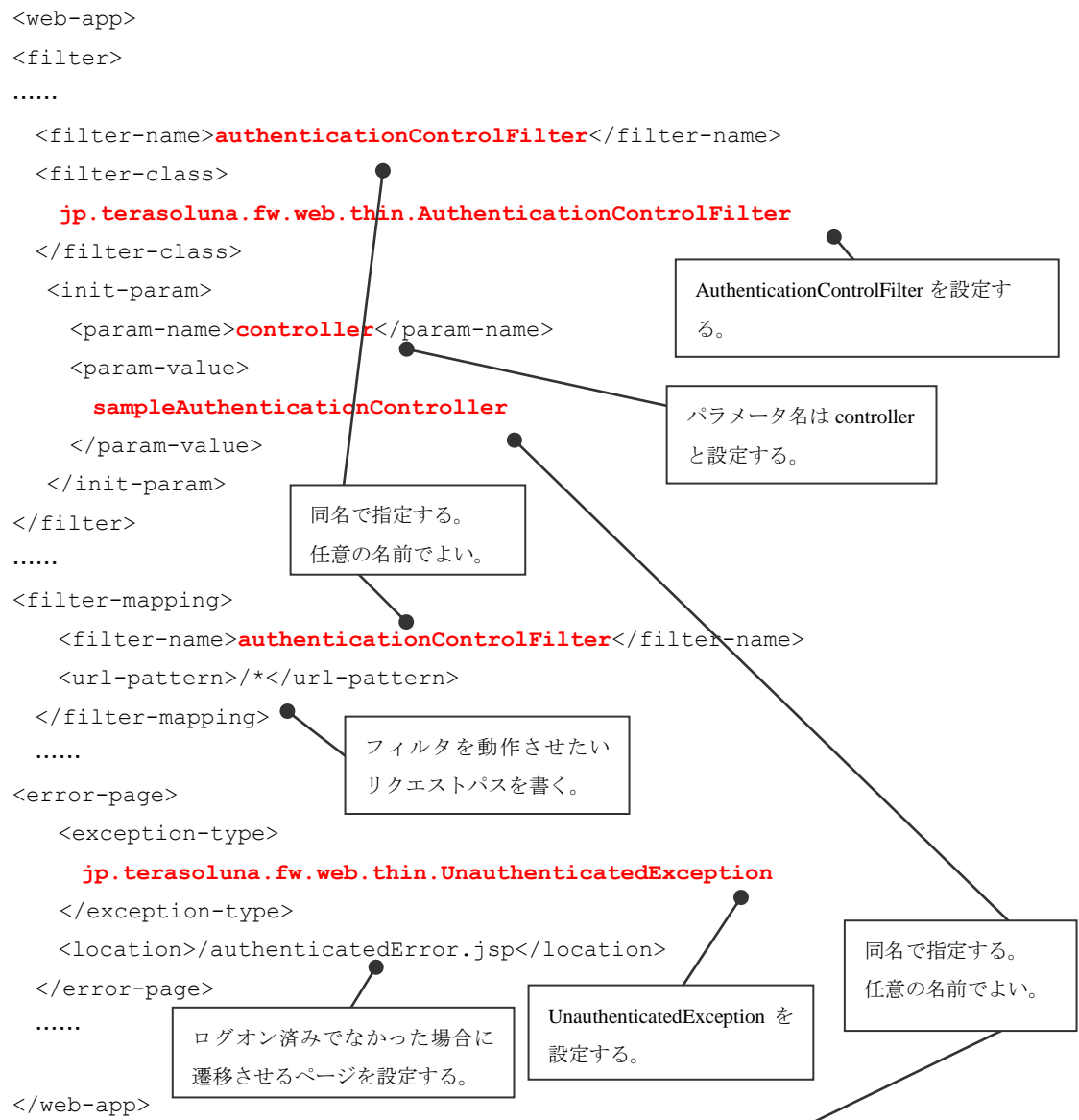
```
<beans>
.....
<bean id="authenticationController"
      class="jp.terasoluna.sample.SampleAuthenticationController"/>
.....
</beans>
```

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

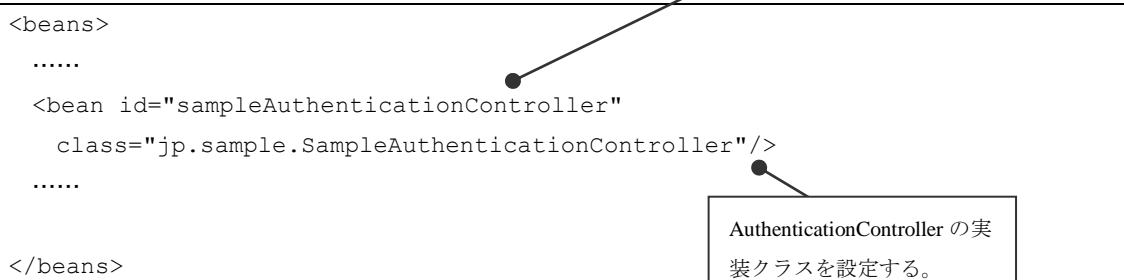
AuthenticationController の実装クラスを設定する。

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)



● bean 定義ファイル



※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

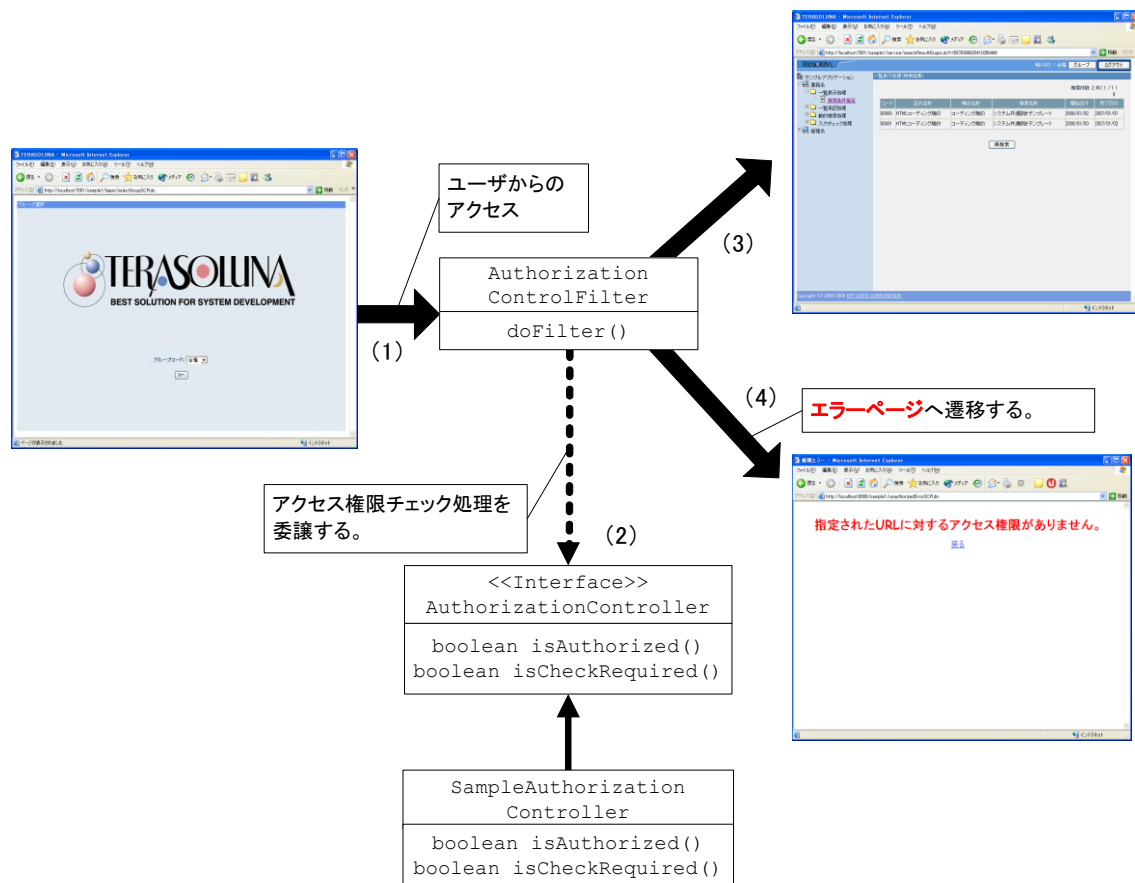
WA-02 アクセス権限チェック機能

■ 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 `AuthorizationControllerFilter` 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

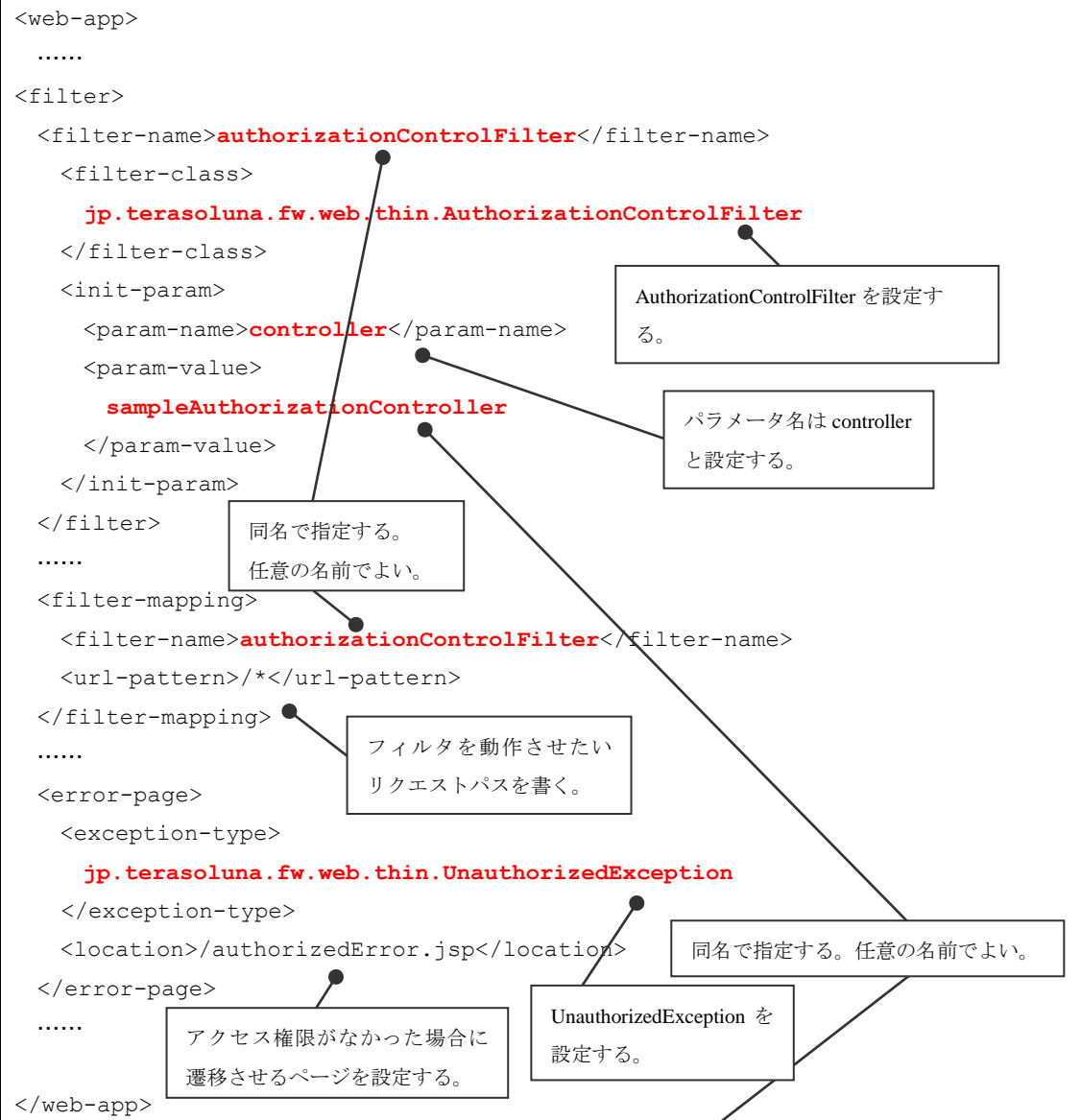
```
<beans>
.....
<bean id="authorizationController"
  class="jp.terasoluna.sample.SampleAuthorizationController"/>
.....
</beans>
```

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

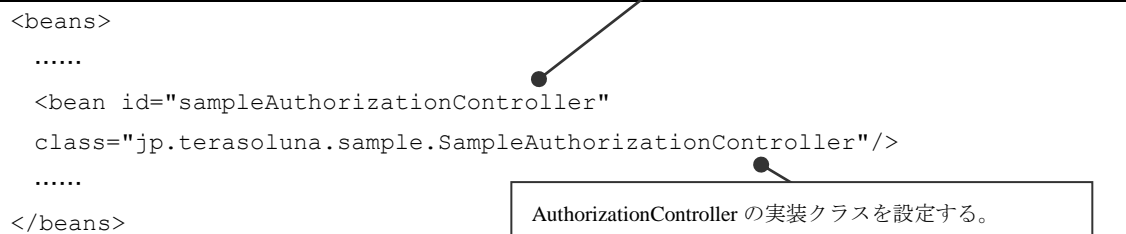
AuthorizationController の実装クラスを設定する。

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)



● bean 定義ファイル



※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	AuthorizationControlFilter 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

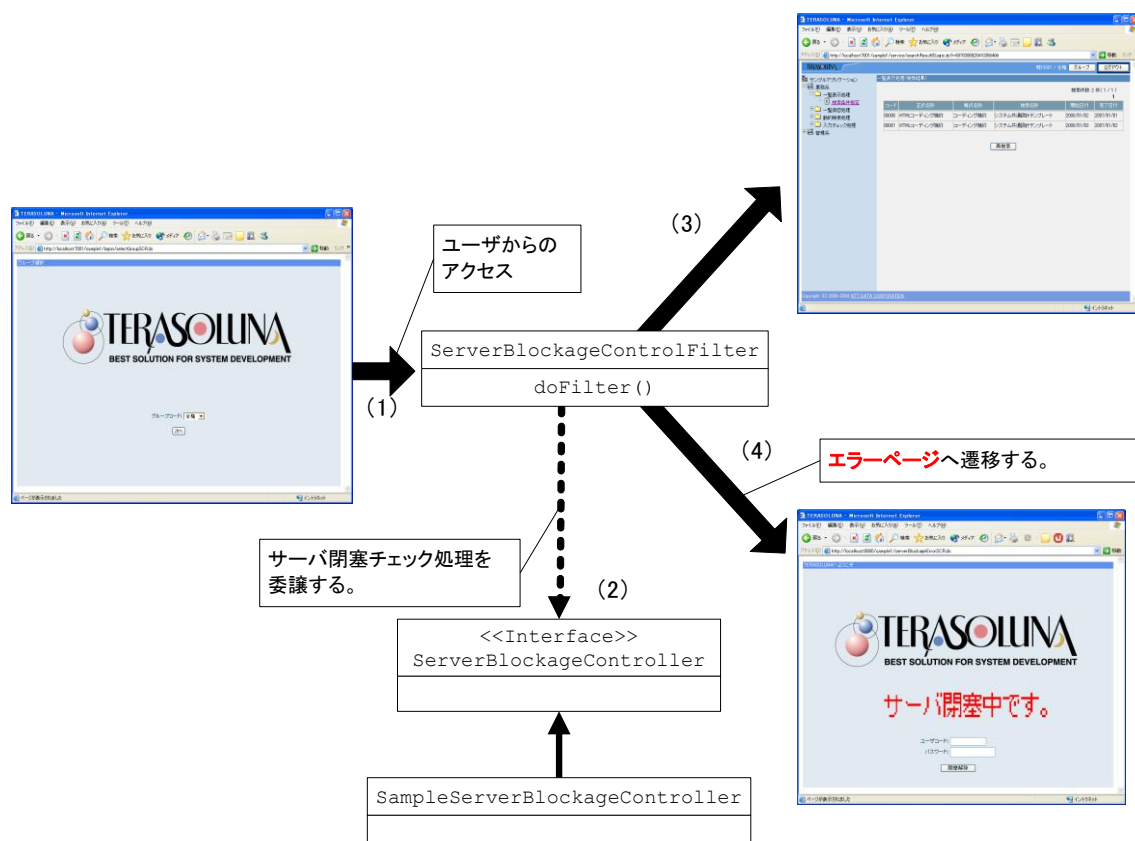
WA-03 サーバ閉塞チェック機能

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

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。

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。

※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

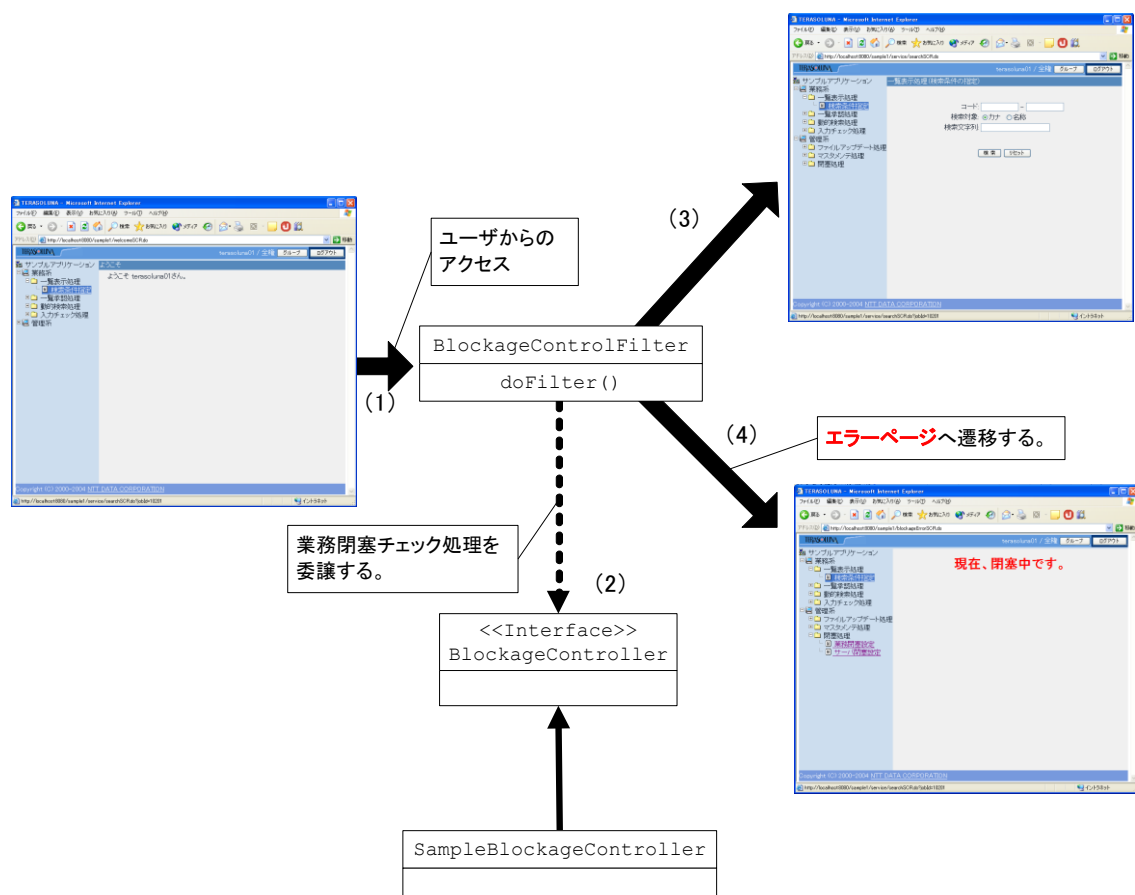
■ 概要

◆ 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

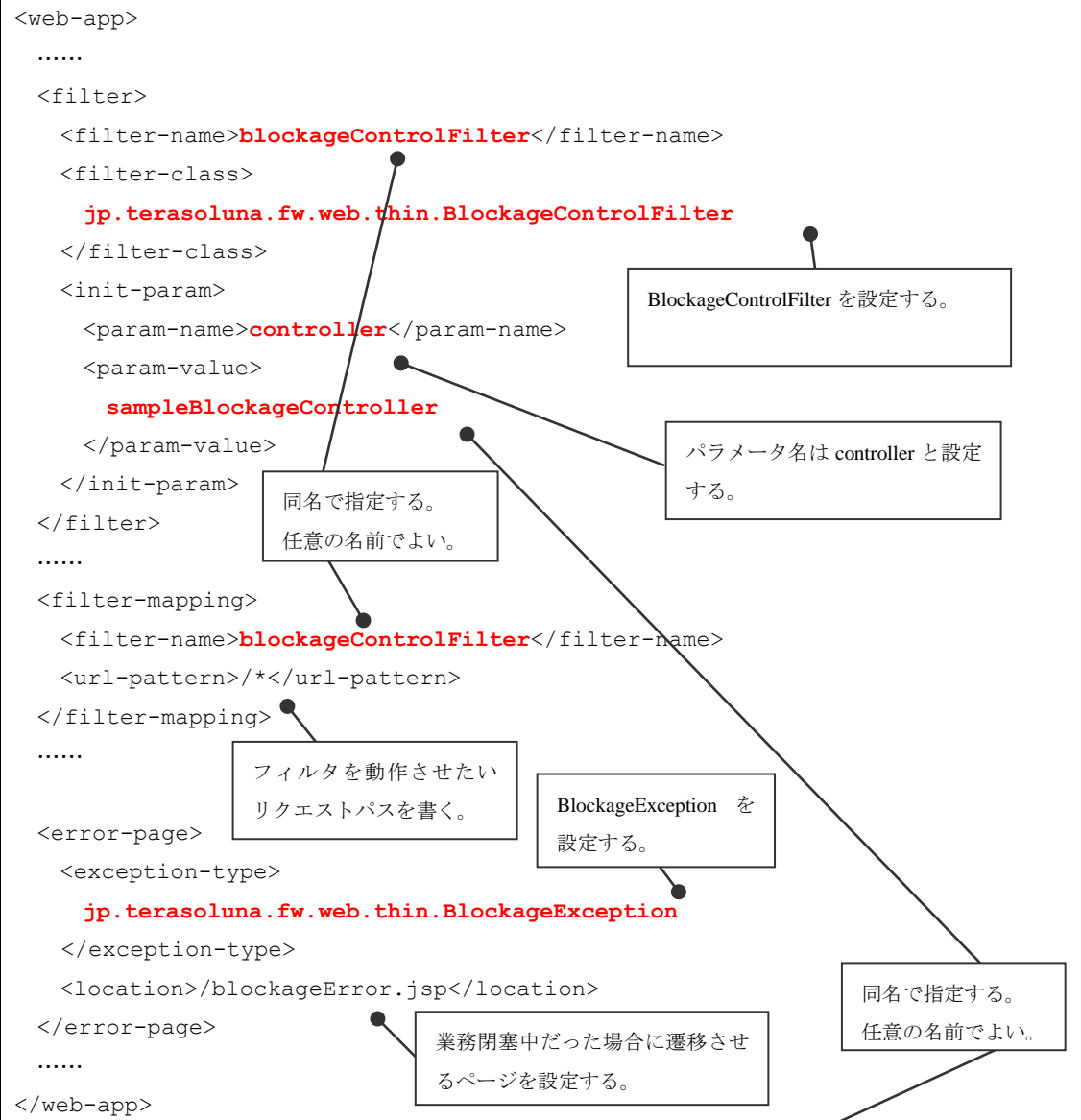
```
<beans>
.....
  <bean id="blockageController"
    class="jp.terasoluna.sample.SampleBlockageController"/>
.....
</beans>
```

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

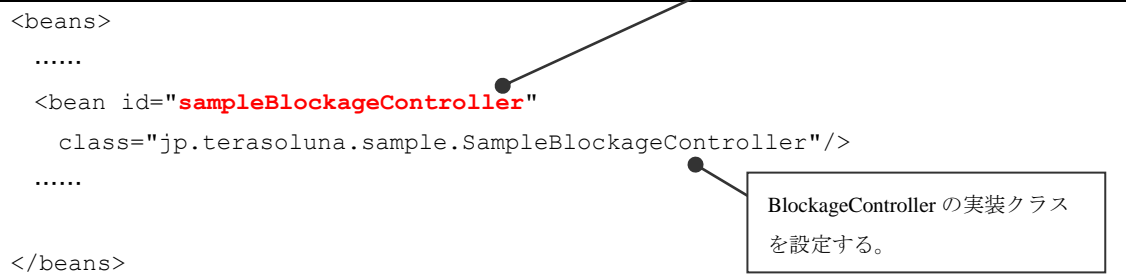
BlockageController の実装クラスを設定する。

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)



● bean 定義ファイル



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

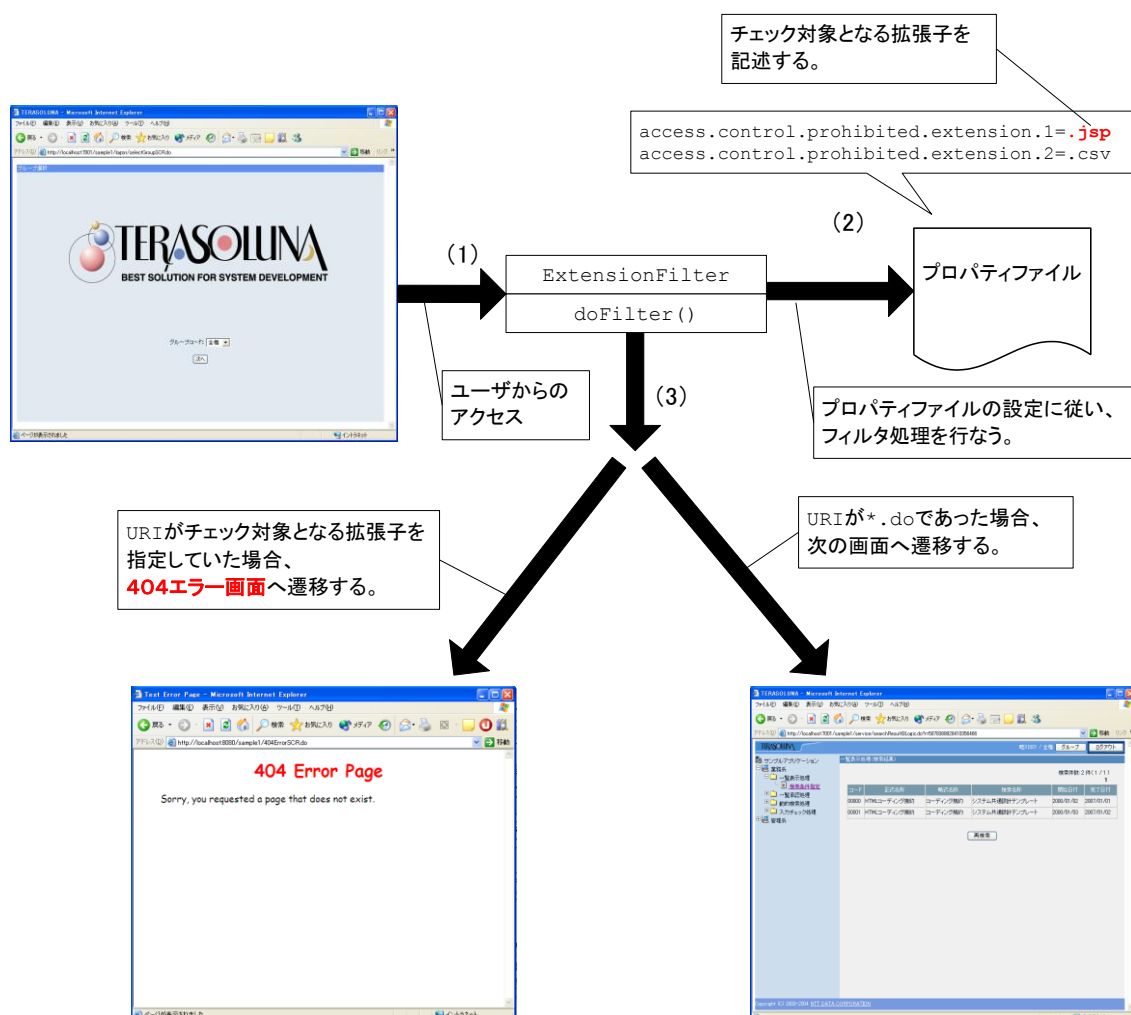
WA-05 拡張子直接アクセス禁止機能

■ 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。

WA-06 セッション同期化機能

■ 概要

◆ 機能概要

- 同一セッションのリクエストが同時に処理されないよう、セッション ID で同期化する。
 - 同期化せずに、セッションスコープのアクションフォームを使用した場合、同一のアクションフォームに対し、複数のスレッドからアクセスできる。この場合、入力チェック後に別スレッドから値が書き換えられてから **Action** や **BLogic** が実行されてしまう等、意図せぬ動作となることがある。これを防ぐために、同期化を行う。
 - セッション ID が同一のリクエストの処理は、同時には1つしか動作しないようにし、セッション ID が異なるリクエストの処理は、それぞれ同時に動作可能とする。
- 同期化によるロック待ちスレッドが溜まった際、古いロック待ちスレッドをロック待ち状態から解放する(ロック待ちを中断させる)。
 - 一部の業務 AP にて、レスポンスが極端に遅いことがあると、レスポンスを返すまでの長い間、そのセッションのリクエストは全て待たされる。その間、ユーザがブラウザにてリロードを繰り返し試行する（レスポンスが無いので、少し前の画面をリロードして、操作をやり直そうとする。これもまたレスポンスが無いので、また同様のことを行う）可能性があり、この場合、一部のユーザの操作によって、サーバのスレッドが占有されてしまう恐れがある。そのため、この同期化機能には、古いロック待ちスレッド（＝リロード操作により、ブラウザがもうレスポンスを必要としていないであろうリクエストを処理しようとしてロックを待っているスレッド）のロック待ちを中断させる機能を実装している。
 - 既にロックを取得し、フィルタを通過しているスレッドを中断させることは無い。
 - 設定により、ロック待ちを一切中断させないこともできる。

◆ 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 `SessionLockControlFilter` 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。

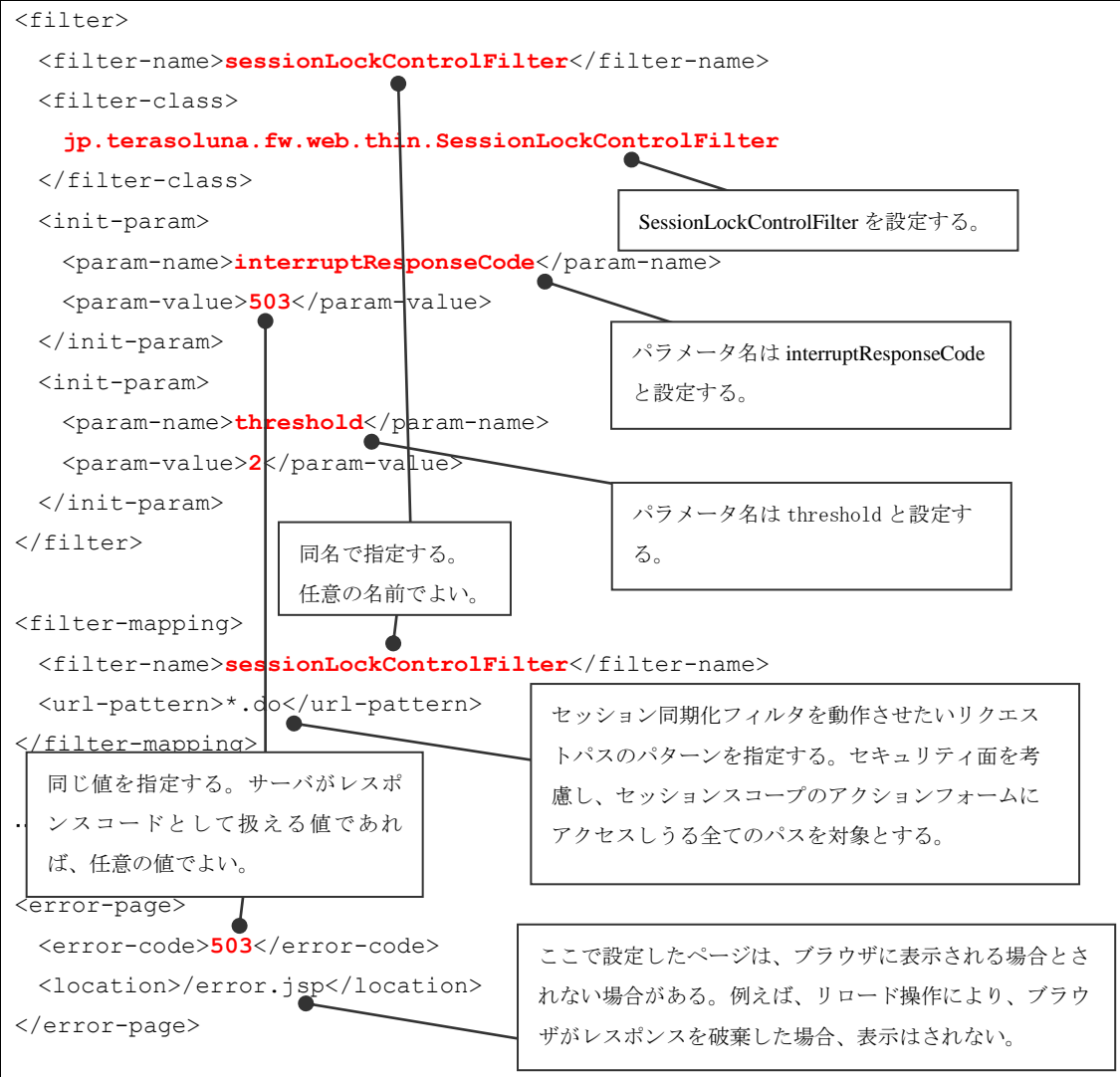
- (1) WebAP コンテナがユーザからのリクエストを受け、`SessionLockControlFilter` に処理を委譲する。
- (2) セッションがつくられている場合、セッション ID で同期化し、後続のリクエスト処理に委譲する。
- (3) 多数のスレッドが、同一のセッション ID のロック待ち状態であった場合、他に比べて早くロック待ち状態となったスレッドから、ロック待ちを中断させる（同一ユーザによるリロード連発時の、スレッド占有回避）。
- (4) ロック待ちが中断された場合、503 エラーレスポンス(レスポンスコード 503、デプロイメントディスクリプタ（web.xml）の `error-page` 要素にエラーページが設定されている場合は、エラーページ）を返す。

■ 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。同ファイルに、同一セッション ID のロック待ちの中断を実行するためのしきい値や、ロック待ち中断時のレスポンスコードを設定できる。書き方については以下の例を参考のこと。

● デプロイメントディスクリプタ (web.xml)



※パラメータ interruptResponseCode (デフォルト 503) と、threshold (デフォルト 2) は、デフォルト値から変更がなければ、init-param 要素を省略してもよい。

● パラメータ interruptResponseCode

- ロック待ちを中断したときのレスポンスコード。
- 同一のコードを error-page 要素内の error-code で指定すると、エラーページを割り当てることが出来る。
 - ◇ エラーページを割り当てていても、レスポンスを中継する Web サーバ (HTTP サーバ) の設定によっては、Web サーバにて別のエラーページに置き換えられる場合がある。

● パラメータ threshold

- ロック待ちを中断させるか否かの判定時に用いるしきい値。
- セッション ID で同期化する際、同一のセッション ID のロックを待つスレッドの数が、この値を既に超えていた場合、より早くロック待ちしているスレッドの、ロック待ちを中断させてから、自身のスレッドがロック待ちに入る。
- 負の値 (「-1」等) を設定すると、ロック待ちの中断は行わない。

◆ 拡張ポイント

特大サイズのファイルをダウンロードさせる業務 AP のように、あらかじめレスポンスに時間がかかることが分かっており、かつ、その時間がかかっている間に、セッションやアクションフォームが書き換えられても処理に影響を与えない作りの業務 AP がある場合、以下のような拡張を行うことで、フィルタ以外から（処理の途中で）セッション ID による同期化のロックを解放する（次のリクエストを処理可能にする）ことができる。

- SessionLockControlFilter の lockLimitedLock メソッドをオーバーライドし、スーパークラスの lockLimitedLock メソッドを実行してから、引数の LimitedLock オブジェクトをリクエスト属性かスレッドローカルに保持する。
- LimitedLock オブジェクトをスレッドローカルに保持した場合は、unlockLimitedLock も併せてオーバーライドし、スーパークラスの unlockLimitedLock メソッドを実行してから、スレッドローカルから LimitedLock オブジェクトを削除する。
- セッション ID による同期化のロックを解放したいタイミングで、リクエスト属性かスレッドローカルに保持された LimitedLock オブジェクトの unlock メソッドを実行する。

この方法は、パラメータ threshold が 0 以上の場合のみ使用できる。ロック待ちの中断を一切させたくなく、かつ、上記の拡張方法を利用する場合は、パラメータ threshold に、AP サーバのスレッド数よりも大きな値(「1000」等)を設定する。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.thin.SessionLockControlFilter	同一セッションの処理の同期化を行うフィルタクラス。
2	jp.terasoluna.fw.web.thin.LimitedLock	ロック待ちスレッドが増えた際に、古いロック待ちスレッドを中断する機能を持つロッククラス。
3	jp.terasoluna.fw.web.thin.SessionLockReference	WeakReference の拡張クラス。

■ 関連機能

- なし。

■ 使用例

- Terasoluna Server Framework for Java (Web 版) ブランクプロジェクト
 - 「セッション同期化」
 - ☆ /webapps/WEB-INF/web.xml

■ 備考

- なし。

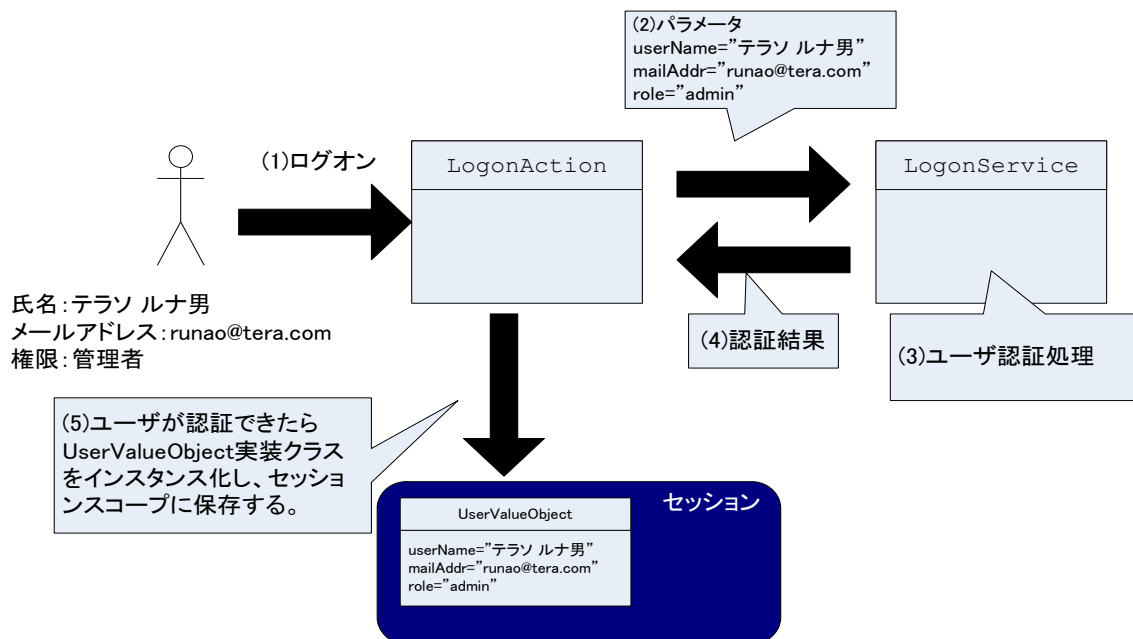
WB-01 ユーザ情報保持機能

■ 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。

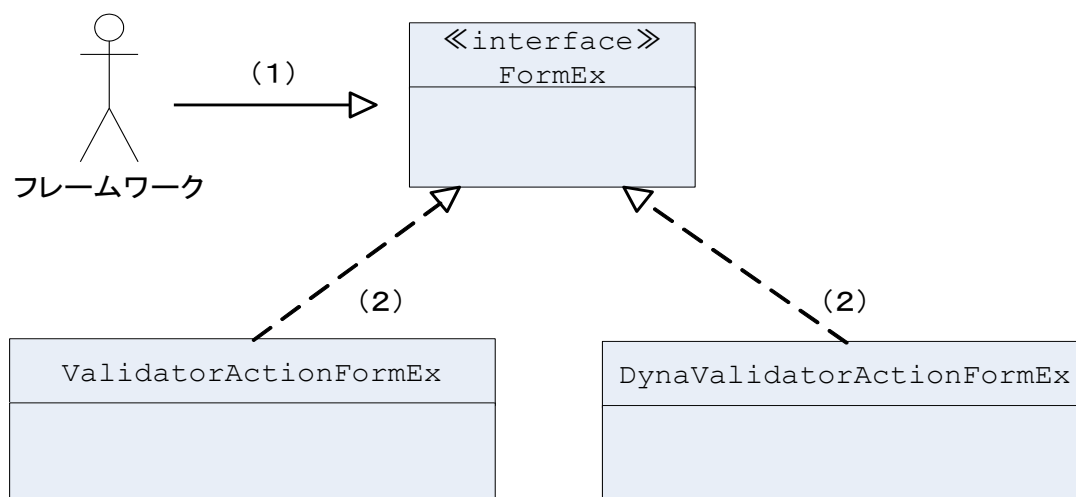
WB-02 アクションフォーム拡張機能

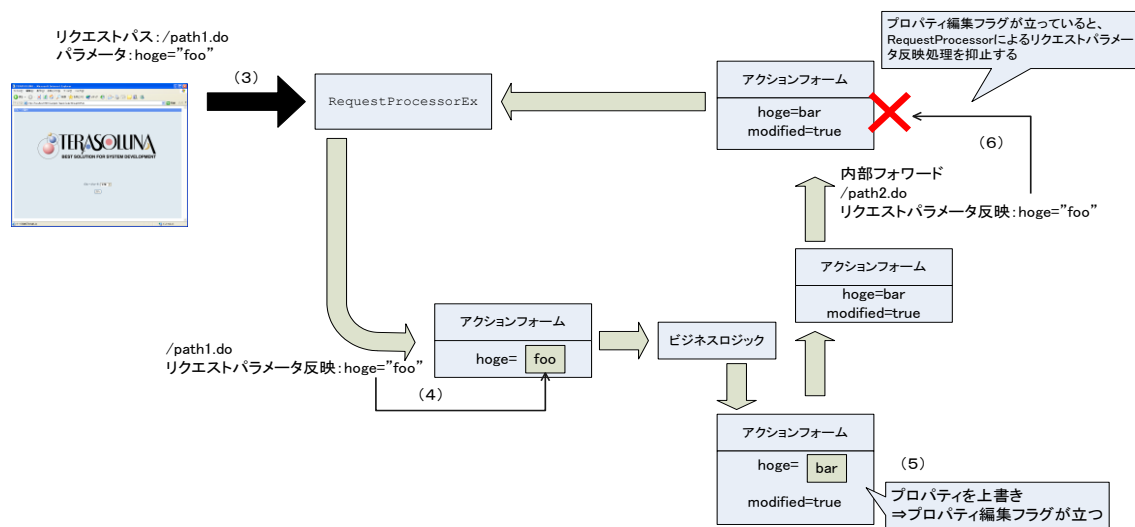
■ 概要

◆ 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

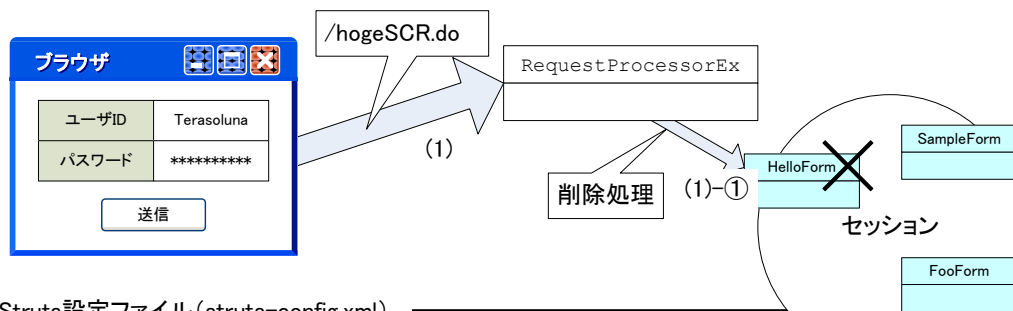
■ 概要

◆ 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、業務毎にアクションフォームを定義することで、同一業務内の操作であれば同じアクションフォームを使用し、別の業務に遷移したら、前の業務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業務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

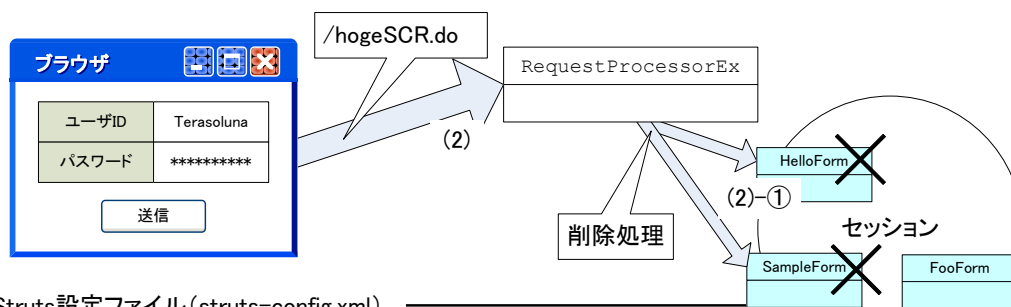
“_”から始まる論理名をもつActionFormに紐づいたパスにアクセスすると・・・



Struts設定ファイル(struts-config.xml)

```
<form-beans>
  <form-bean name="_sampleForm" type="jp.co.nttdata.sample.web.form.SampleForm" />
  <form-bean name="_helloForm" type="jp.co.nttdata.sample.web.form.HelloForm" />
  <form-bean name="fooForm" type="jp.co.nttdata.sample.web.form.FooForm" />
</form-beans>
...
<action-mapping type="jp.co.nttdata.terasoluna.fw.web.struts.action.ActionMappingEx">
  <action path="/hogeSCR"
    type="jp.co.nttdata.terasoluna.fw.web.struts.actions.ForwardAction"
    parameter="/sample/hoge.jsp"
    name="_sampleForm"
    scope="session" />
  </action>
</action-mapping>
```

“_”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・・・



Struts設定ファイル(struts-config.xml)

```
<form-beans>
  <form-bean name="_sampleForm" type="jp.co.nttdata.sample.web.form.SampleForm" />
  <form-bean name="_helloForm" type="jp.co.nttdata.sample.web.form.HelloForm" />
  <form-bean name="fooForm" type="jp.co.nttdata.sample.web.form.FooForm" />
</form-beans>
...
<action-mapping type="jp.co.nttdata.terasoluna.fw.web.struts.action.ActionMappingEx">
  <action path="/hogeSCR"
    type="jp.co.nttdata.terasoluna.fw.web.struts.actions.ForwardAction"
    parameter="/sample/hoge.jsp"
    name="_sampleForm"
    scope="session" />
    <set-property property="clearForm" value="true" />
  </action>
</action-mapping>
```


◆ 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。

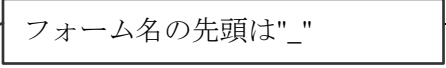
■ 使用方法

◆ 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"/>
```



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

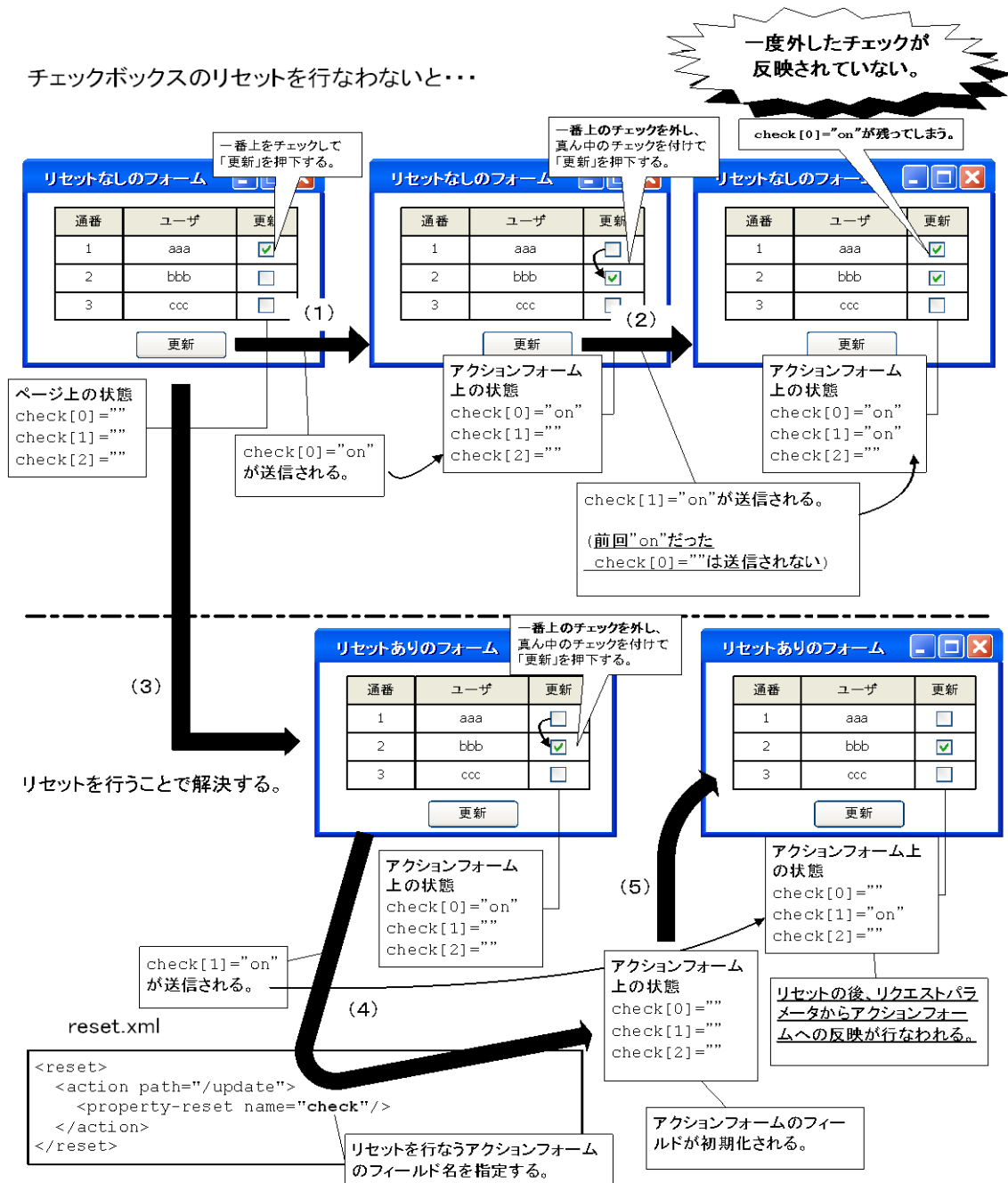
◆ 機能概要

- 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** クラスでは配列や **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。
利用例としてページリンクを伴う一覧表示にチェックボックスが存在する場合に、現在表示しているページのチェックボックスのプロパティ値のみを初期化するなどが挙げられる。
- **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に使用する際は **BLogicAction** のアクションパスでリセット処理を設定することを推奨する。**BLogicAction** 以外のアクションパスでリセット処理を設定すると、TERASOLUNA のアクションフォーム拡張機能により、意図せぬ場面でリセット処理される、もしくはリセット処理されない可能性がある。確実にプロパティ値を初期化したい場合、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) 【リセットを行なわない場合】

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】

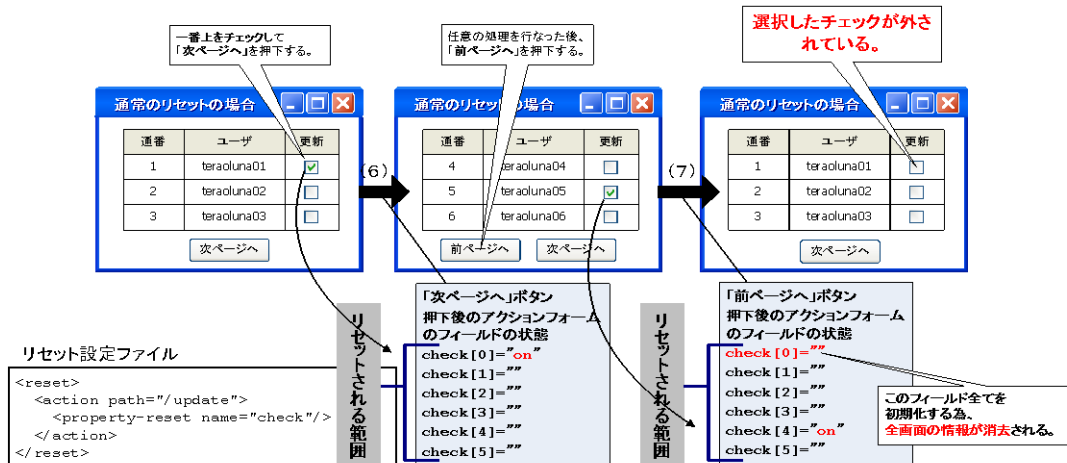
アクションフォームのリセット機能を、リセット定義ファイル (`reset.xml`) ・ Struts 設定ファイル (`struts-config.xml`) で定義してサブミットを行なう。

(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。

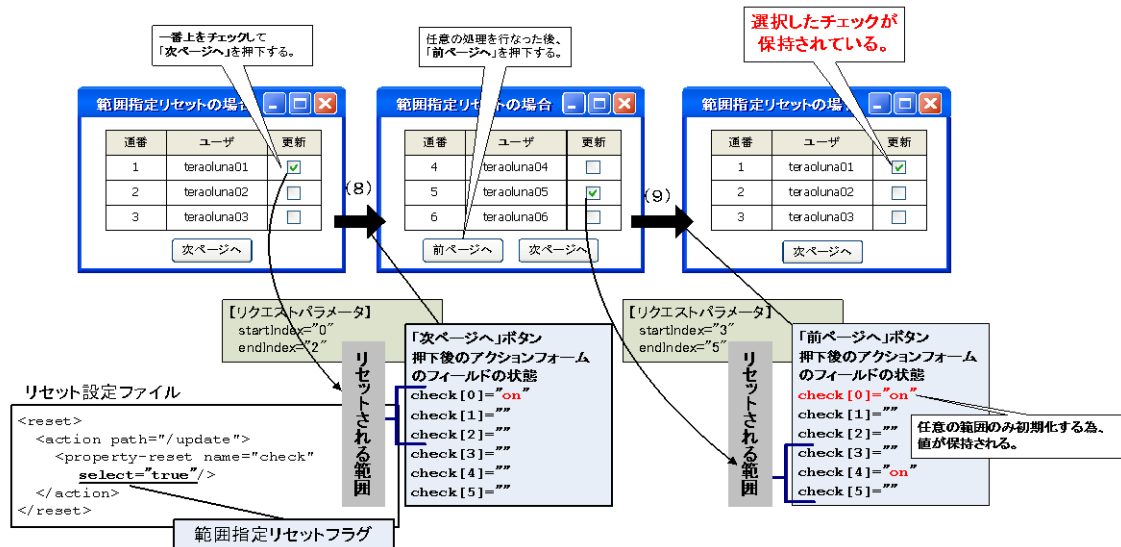
(5) リセット後のアクションフォームフィールドに、リクエストパラメータで送信された値が格納される。これにより、直前の画面 (フォーム) の状態がアクションフォームに登録され、リクエストをまたがった不要な `check[0]="on"` の設定を消去することができる。

◆ 概念図

下記は ResetterImpl クラスを使用した場合の指定範囲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一有効）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) 指定範囲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。
XPathIndexedBeanWrapper の仕様に従い、ネストしたプロパティを記述可能。

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

指定範囲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。

- 指定範囲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。

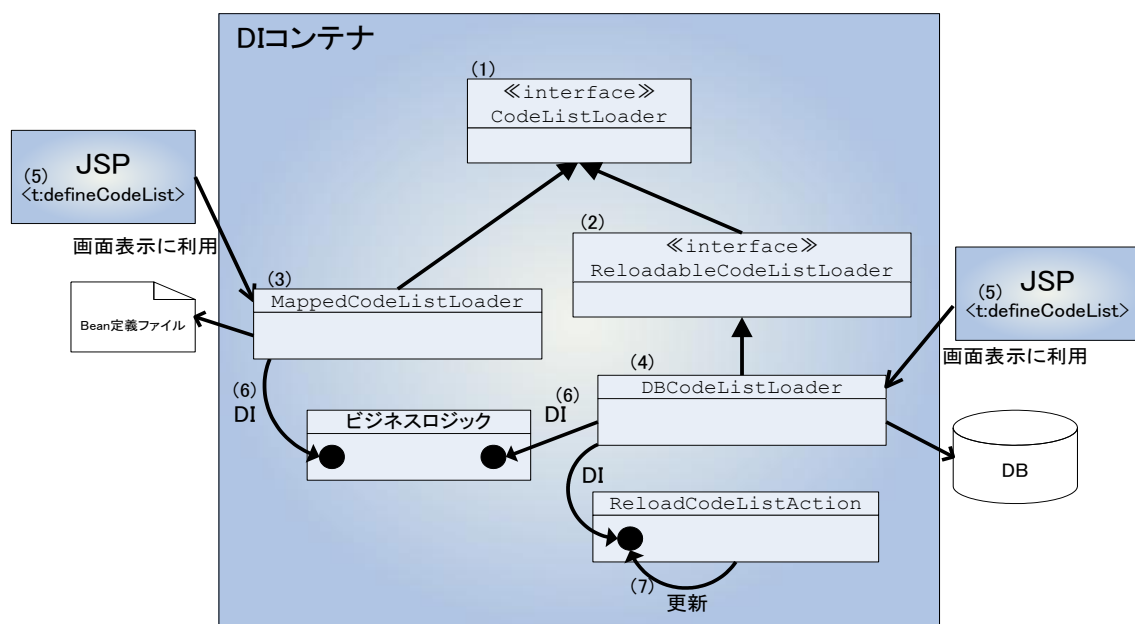
WB-05 コードリスト機能

■ 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。なお、コードリストの順序は Bean 定義ファイルの記述順となる

【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="TerasolunaDataSource"/>
  <property name="sql">
    <value>SELECT KEY, VALUE FROM CODE_LIST ORDER BY KEY</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。なお、コードリストの順序は DB から読み込まれた順となる。コードリストのキー順にしたい場合、ORDER BY 句を利用すること。

【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"/>
```

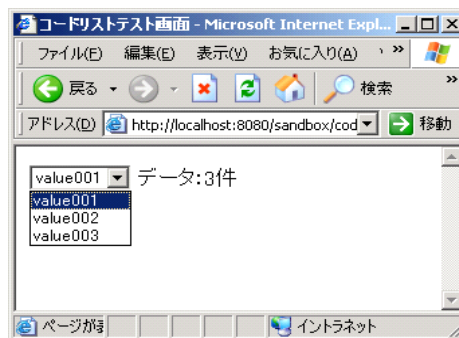
```
</html:select>
```

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

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

コードリストを page スcope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="sampleCodeList"/>
</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="sampleDBCodeList"/>
</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.*

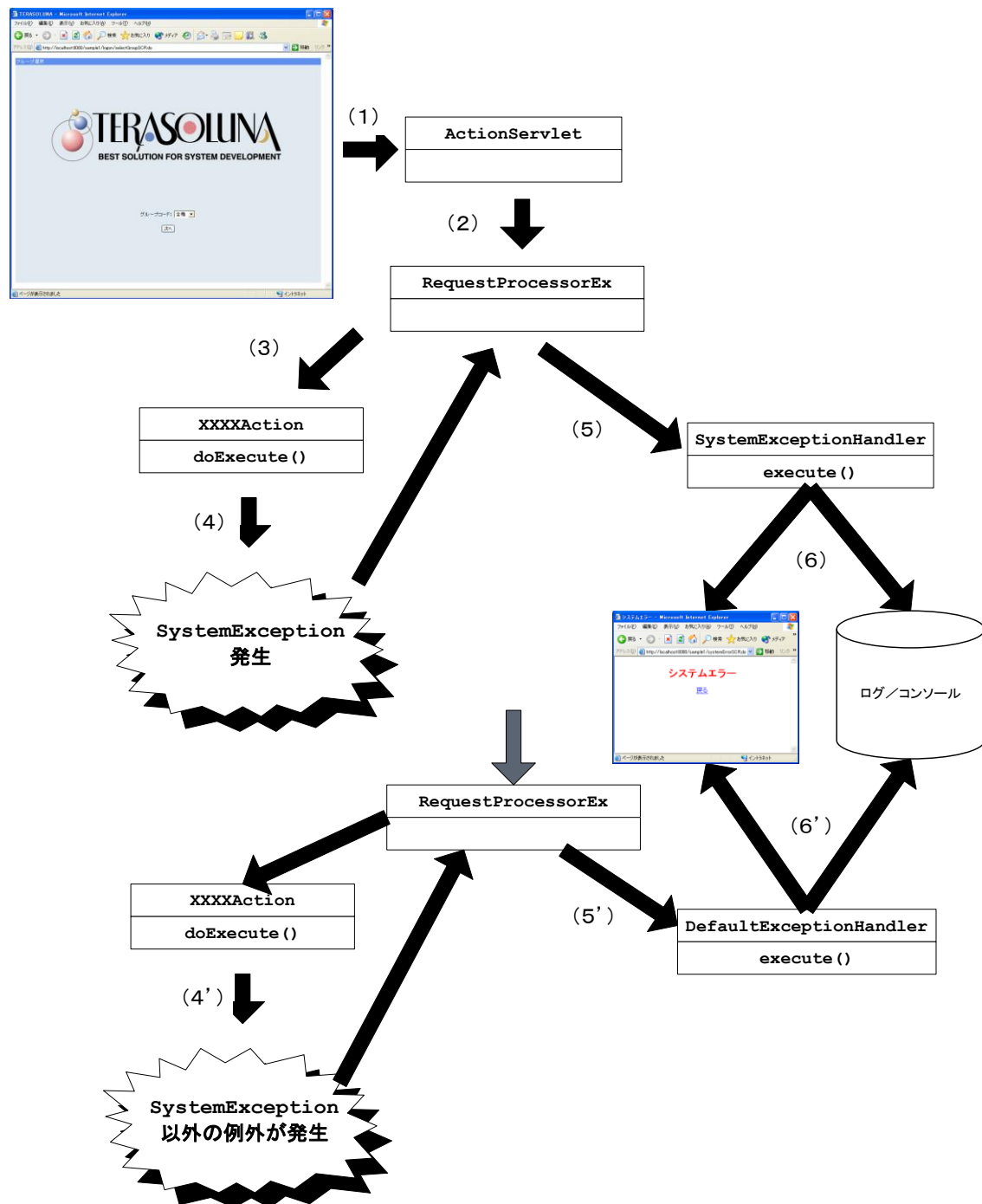
WC-01 例外ハンドリング機能

■ 概要

◆ 機能概要

- Action 実行時（ビジネスロジックを含む）に例外が発生した場合に、共通的な例外処理機能を提供する。
 - **SystemExceptionHandler**
 - ◇ SystemException 専用の例外処理ハンドラ
 - ◇ 置換文字列が指定されていた場合、エラーメッセージへ埋め込む
 - ◇ ログレベルの設定が可能
 - **DefaultExceptionHandler**
 - ◇ 汎用例外ハンドラ
 - ◇ ログレベルの設定が可能
- 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。（ログレベル設定可能）
 - 例外発生時のエラー画面に遷移させる。

◆ 概念図



◆ 解説

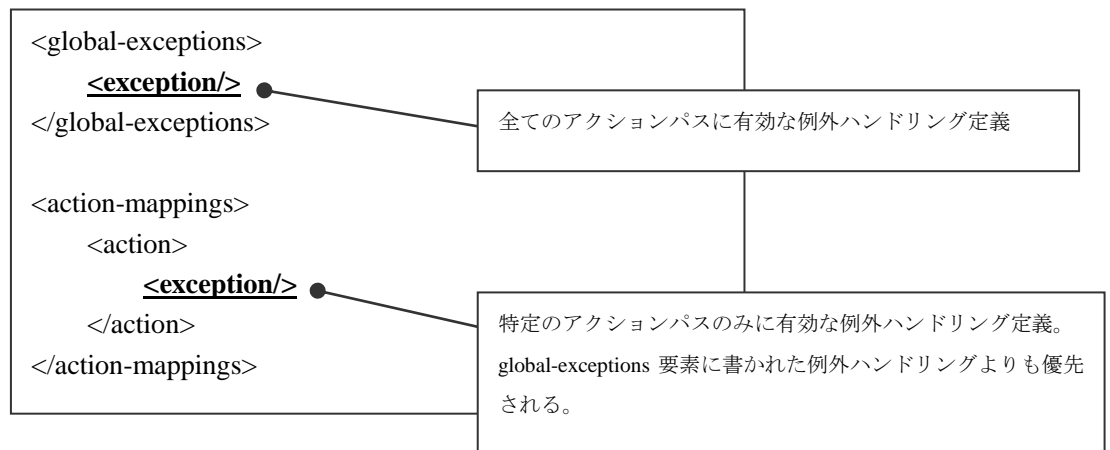
以下の解説は Struts 設定ファイル(struts-config.xml)に `SystemExceptionHandler`、および `DefaultExceptionHandler` の設定が正しくされている事が前提となる。設定方法の詳細は、コーディングポイントを参照のこと。

- (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)。
-
- (4') アクション内部で例外 (システム例外およびその継承クラス以外) が発生する。
 - (5') 例外がキャッチされ、例外を処理するクラス (`DefaultExceptionHandler#execute()`メソッド) が呼び出される。
 - (6') `DefaultExceptionHandler#execute()`メソッドでは、発生した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、`Globals.EXCEPTION_KEY` 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<global-exceptions>要素、あるいは<action>要素に<exception>要素を指定する。



Struts 設定ファイルの global-exceptions 要素(struts-config.xml)

```
<struts-config>
```

```
...
```

<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。

遷移先のパスを指定する。

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、遷移先モジュールとログのエラーレベルを指定できる。

```
<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" />
```

```
    <set-property property="logLevel" value="fatal"/>
```

```
/>
```

type 属性に SystemException を指定する。

ログレベルには trace, debug, info, warn, error, fatal のいずれかを指定する。無指定の場合は error と指定した場合と同等となる。

SystemException あるいはそれを継承した例外クラスのハンドリングには必ず、handler 属性に SystemExceptionHandler、を指定する。

SystemException 以外のハンドリングには handler 属性に DefaultExceptionHandler 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。

```
<exception key="errors.E001" path="/systemError.do"
```

```
  className="jp.terasoluna.fw.web.struts.action.ExceptionConfigEx"
```

```
  handler="jp.terasoluna.fw.web.struts.action.DefaultExceptionHandler"
```

```
  type="java.lang.Exception" >
```

```
    <set-property property="module" value="/sub" />
```

```
    <set-property property="logLevel" value="error"/>
```

```
/>
```

```
</global-exceptions>
```

```
</struts-config>
```

- Struts 設定ファイルの action 要素(struts-config.xml)

```
<struts-config>
. . .
<action-mappings>
  <action-mapping
    type="jp.terasoluna.fw.web.struts.action.ActionMappingEx">

    <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>
    </action-mapping>
  </action-mappings>
</struts-config>
```

<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。

- エラー画面の例（SystemExceptionHandler でハンドリングさせる場合）（JSP）

```
<%@ taglib uri="/terasoluna-struts" prefix="ts"%>
(略)
<%
  java.lang.Exception sysExp =
    (java.lang.Exception) request.getAttribute(PageContext.EXCEPTION);
%>

<ts:errors/>
<%=sysExp.getMessage()%>
(略)
```

SystemExceptionHandler の場合、PageContext.EXCEPTION キーで例外インスタンスが取得できる。

- エラー画面の例（DefaultExceptionHandler でハンドリングさせる場合）（JSP）

```
<%@ taglib uri="/terasoluna-struts" prefix="ts"%>
(略)
<%
    java.lang.Exception exp =
        (java.lang.Exception) request.getAttribute(Globals.EXCEPTION_KEY);
%>

<ts:errors/>
<%=exp.getMessage() %>
(略)
```

DefaultExceptionHandler の場合、
Globals.EXCEPTION_KEY キーで
例外インスタンスが取得できる。

◆ 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.web.struts.action.DefaultExceptionHandler	システム例外以外を処理するハンドラクラス。
3	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

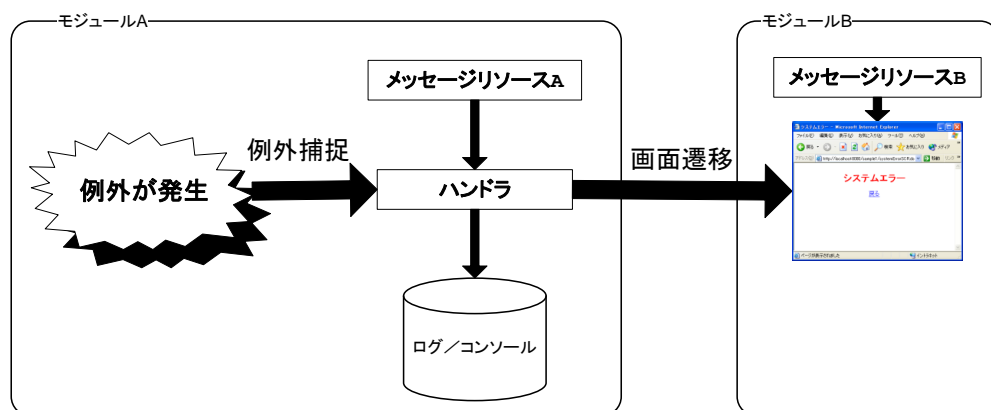
- なし

■ 使用例

- Terasoluna Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC14 例外ハンドリング処理」
 - ◇ /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

- 例外ハンドリング機能で使われるメッセージリソースについて
サブモジュールで発生した例外を例外ハンドラで補足しエラー画面に遷移させるケースで、遷移先のエラー画面が別のモジュールにある場合は、ログのメッセージとエラー画面のメッセージとで参照されるメッセージリソースが変わるので注意が必要である。
この場合、ログ出力のメッセージリソースには元のモジュールのメッセージリソースが使われる。エラー画面のメッセージには遷移先のメッセージリソースが使われる。



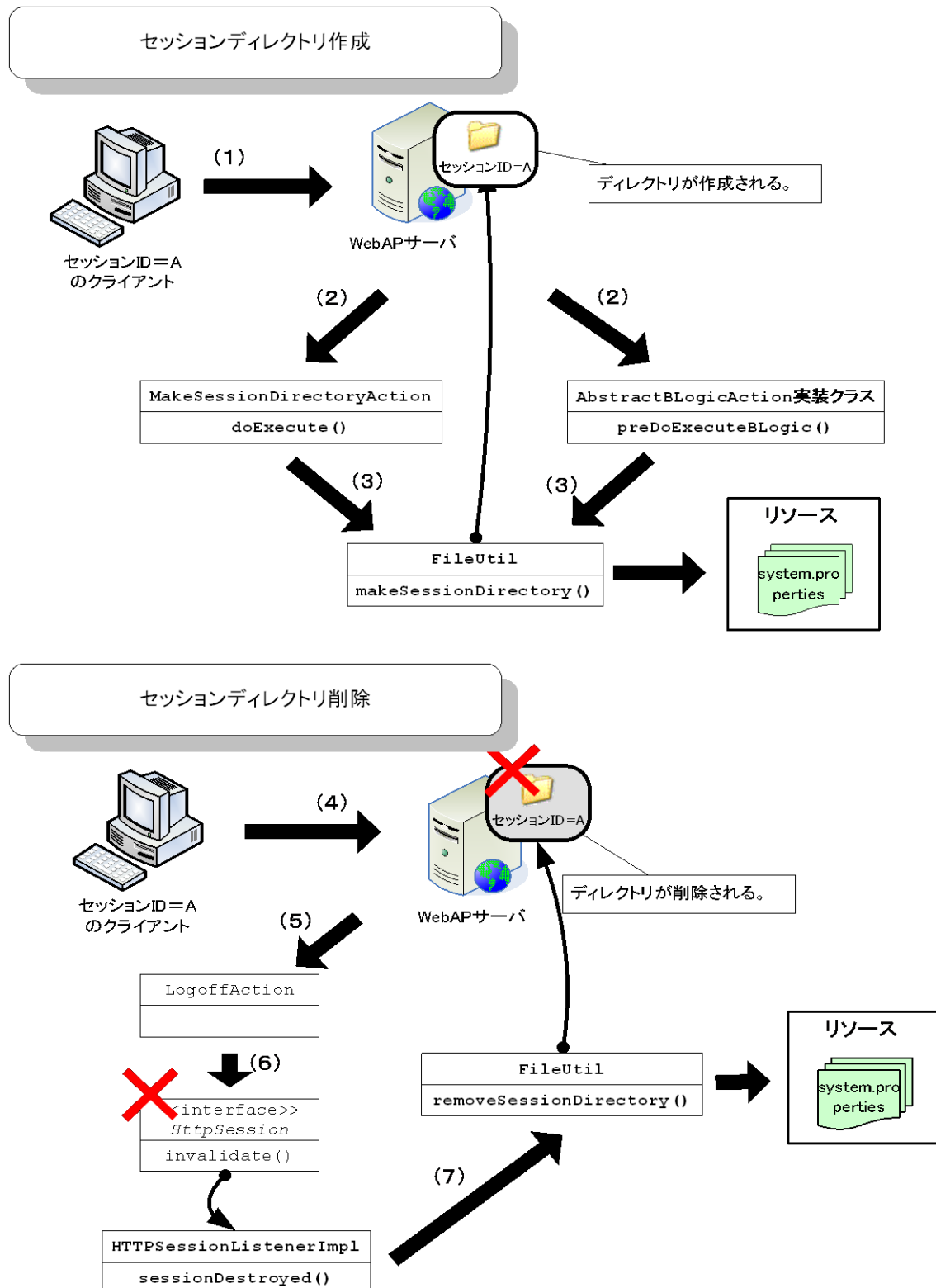
WD-01 セッションディレクトリ機能

■ 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="SampleBLogic"/>  
</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

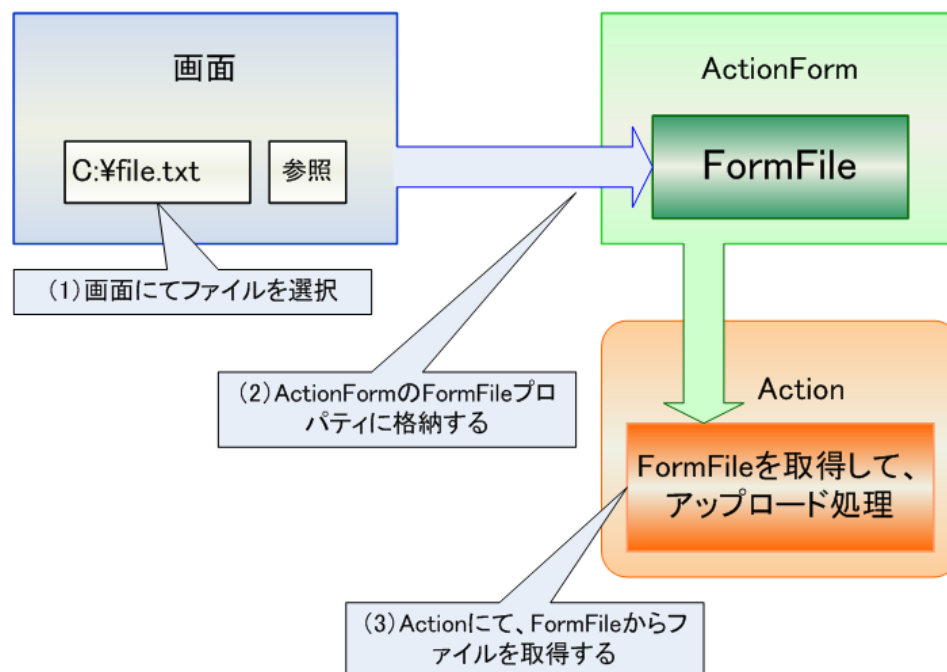
WD-02 ファイルアップロード機能

■ 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。

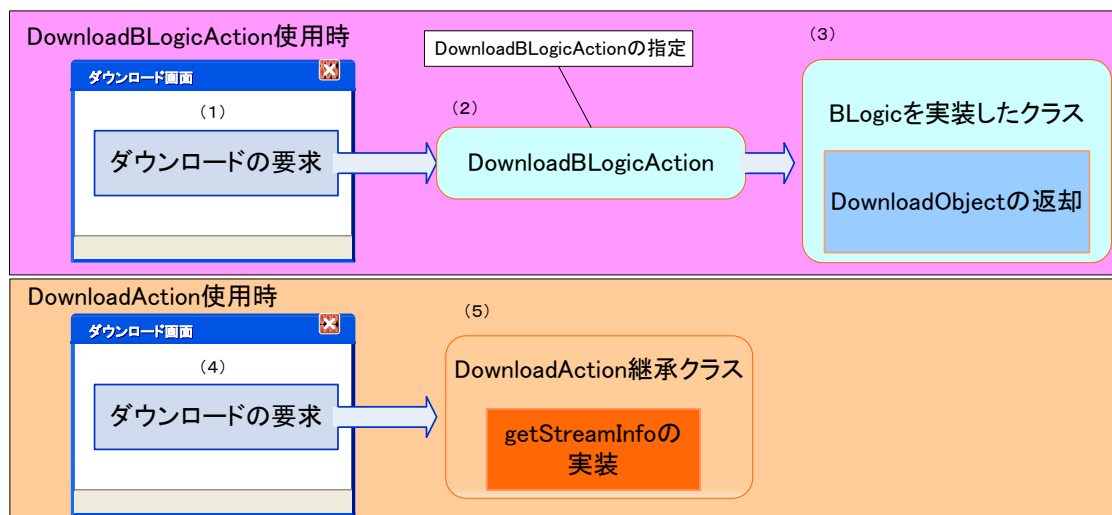
WD-03 ファイルダウンロード機能

■ 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

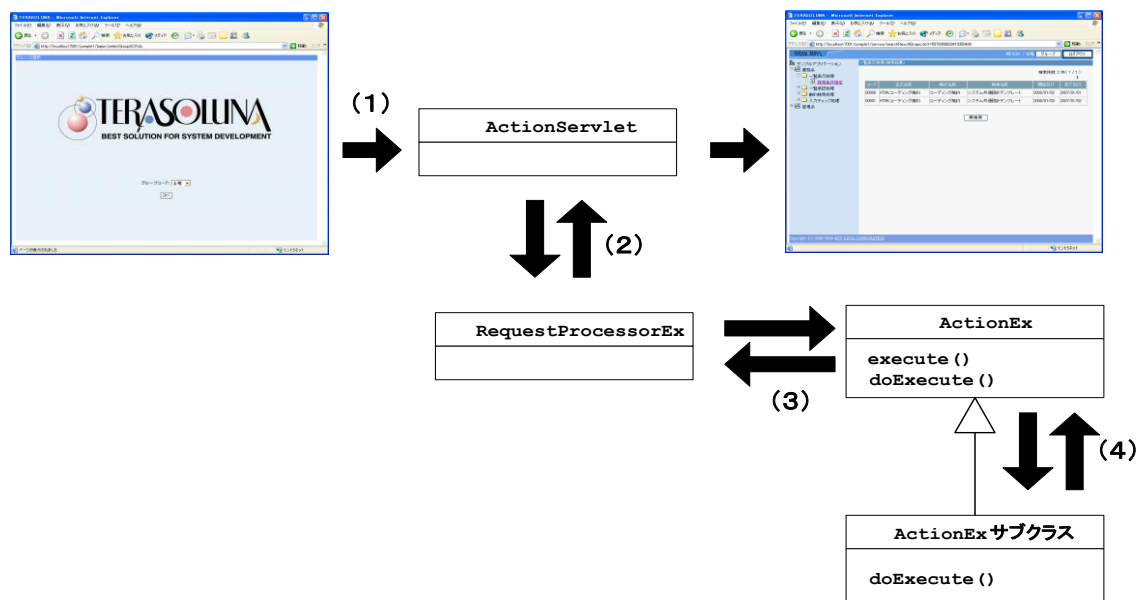
- TERASOLUNA Server Framework for Java (Web 版)の全てのアクションクラスの基底クラスを提供する。
- 以下の機能を提供する。
 - トランザクショントークンチェックを起動する。
 - ビジネスロジックの実行後等で保存先がセッションに指定されているエラー情報、メッセージ情報をセッションスコープに設定する。

※ 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

- アクションクラスの Bean 定義の scope について
 - アクションクラスの Bean 定義について、scope="prototype" もしくは scope="singleton"のいずれかを指定する。
 - ✧ "singleton"を選択した場合は、アクションクラスをマルチスレッドセーフに実装する必要がある。"prototype"を指定した場合はその必要はないが、毎回インスタンスを生成するため、性能劣化に注意する必要がある。
- 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"/>
    <property name="saveToken" value="false"/>
</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。

➤ トランザクショントークンチェックの設定例

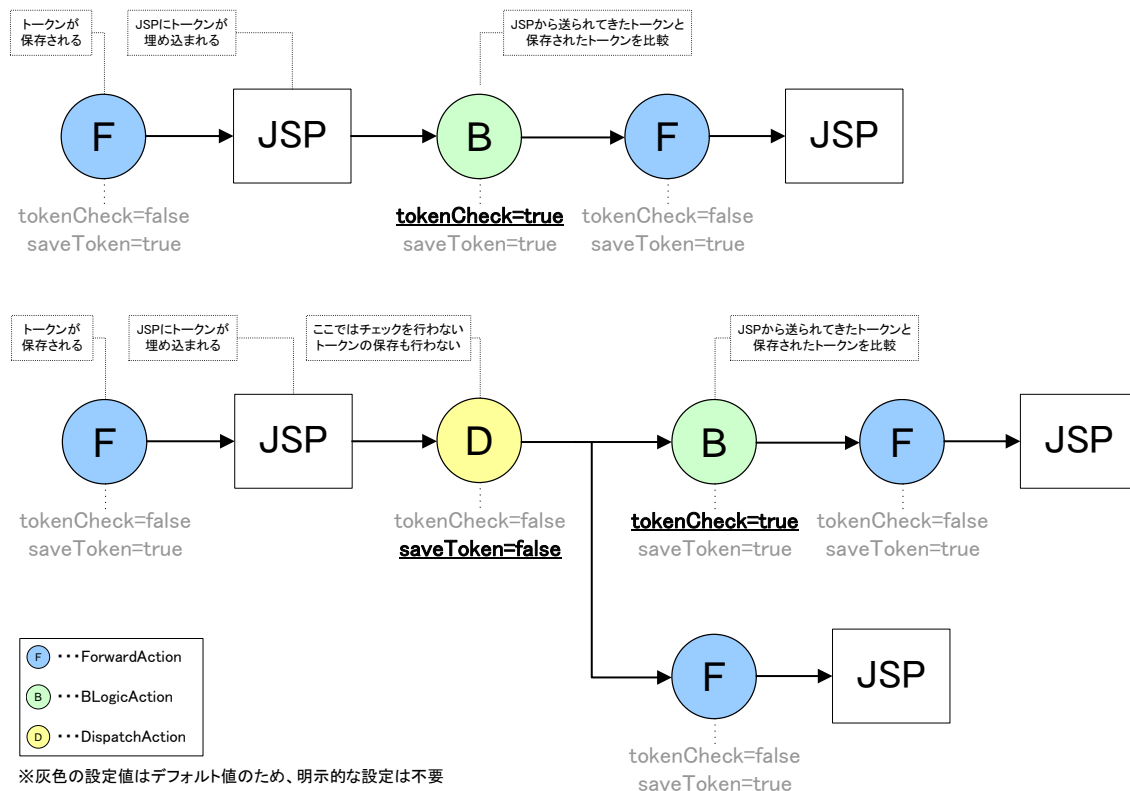
以下の図はトランザクショントークンの設定を表したイメージ図である。

ここでは2つの設定例について紹介する。

一つ目の例は、画面からトークンチェックの必要なビジネスロジックを直接呼び出し、別画面へ遷移するパターンである。BLogicAction の Bean 定義で tokenCheck プロパティを true に設定することで、遷移元画面で保存されたトークンを元にビジネスロジック起動前にトークンチェックを行う。

二つ目の例は、画面で押されたボタンやラジオボタンの選択によって、遷移先を振り分けるパターンである。この場合、DispatchAction ではトークンチェックを行わずに、振り分けた先の BLogicAction でトークンチェックを行う。

注意すべき所は、DispatchAction では saveToken プロパティに false を設定し、トークンが保存されないようにする点である。



※灰色の設定値はデフォルト値のため、明示的な設定は不要

■ 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 コードリスト再読み込み機能』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』
- 『WH-01 ビジネスロジック実行機能』

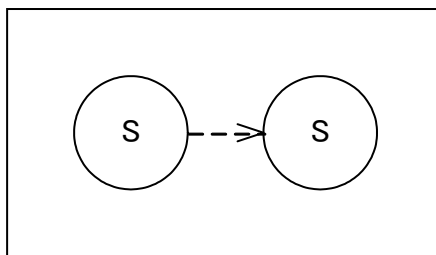
■ 使用例

- 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

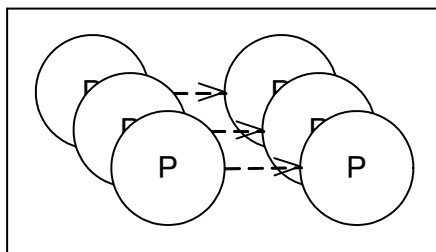
- **prototype の Bean を singleton の Bean にインジェクションする際の留意点**
スコープは個々の Bean 定義に書くが、DI コンテナがそれらを組み合わせたときに、想定とは異なる形になる場合がある。たとえば、singleton スコープで定義されている Bean は実体が 1 つしかないため、そのプロパティに prototype スコープの Bean がインジェクションされていた場合、prototype スコープの Bean も実質的に singleton となる。

- singleton の Bean が singleton の Bean を参照している状態の概念図



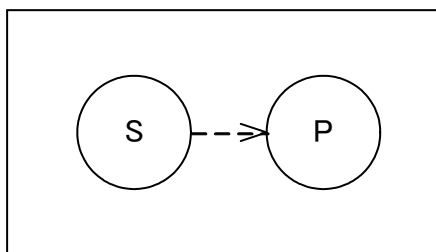
singleton の Bean に singleton の Bean をインジェクションする場合は、組み合わせたものも同じようにオブジェクトはシステムに 1 つだけである。

- prototype の Bean が prototype の Bean を参照している状態の概念図



prototype の Bean に prototype の Bean をインジェクションする場合は、組み合わせたもの全体も毎回新規に生成される。

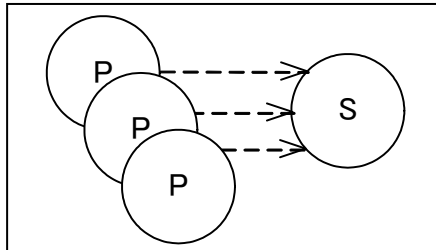
- singleton の Bean が prototype の Bean を参照している状態の概念図



singleton の Bean に prototype の Bean をインジェクションした場合は、singleton スコープには 1 つしかオブジェクトがないため、prototype スコープの Bean も他にインジェクションされていなければ 1 つしか存在しない。この場合、prototype の Bean もスレッドセー

フを意識すること。

- prototype の Bean が singleton の Bean を参照している状態の概念図



逆に prototype の Bean に singleton の Bean をインジェクションした場合は、複数のオブジェクトから同じ singleton のオブジェクトが参照されていることになる。

上記の関係があるため、DI コンテナから最初を取得されるオブジェクトの範囲を意識しておくこと。TERASOLUNA の場合は、最初に Action クラスが 1 リクエスト（大体の AP サーバでは 1 スレッド）ごとに DI コンテナから取得される。

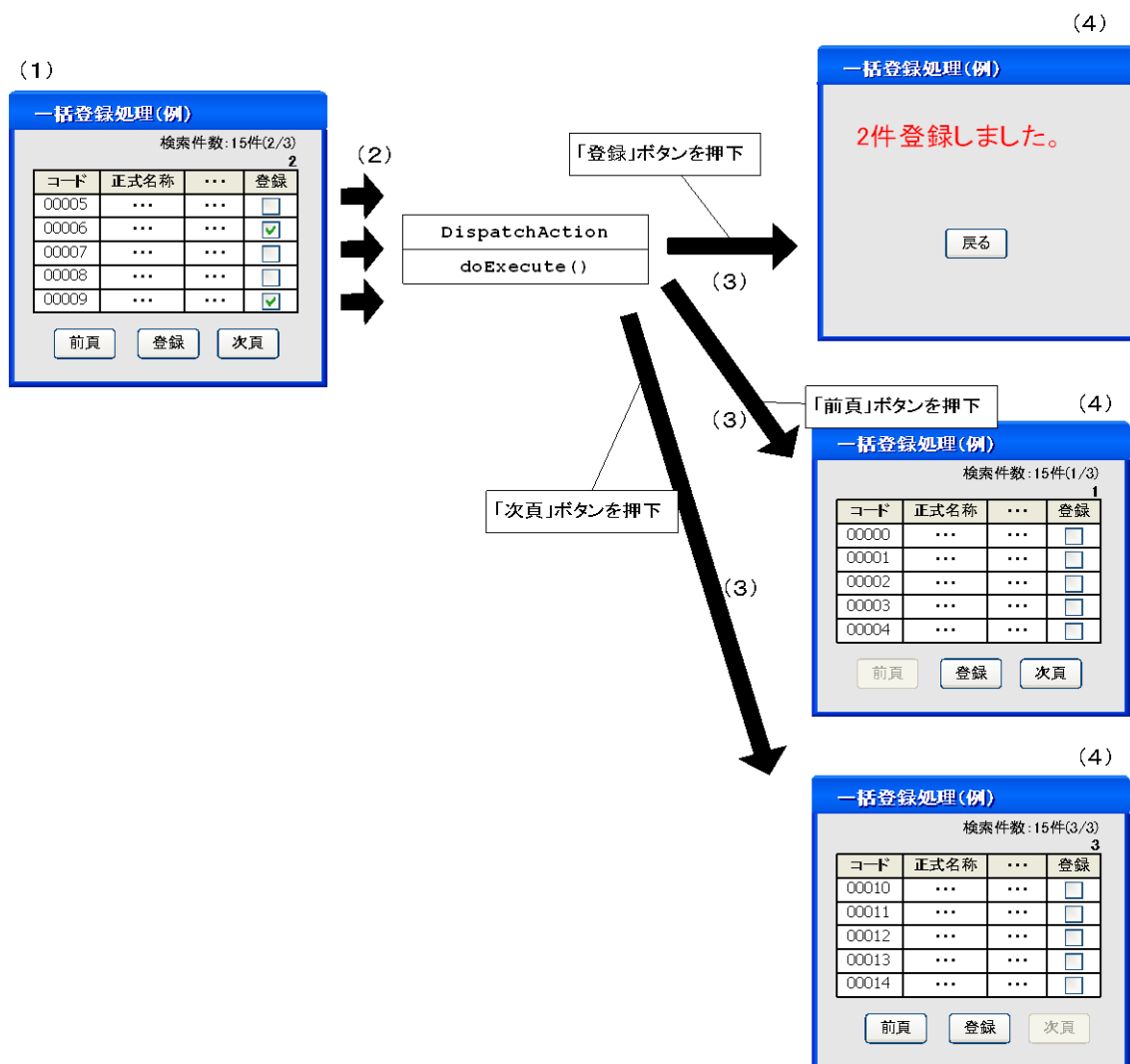
WE-02 標準ディスパッチャ機能

■ 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



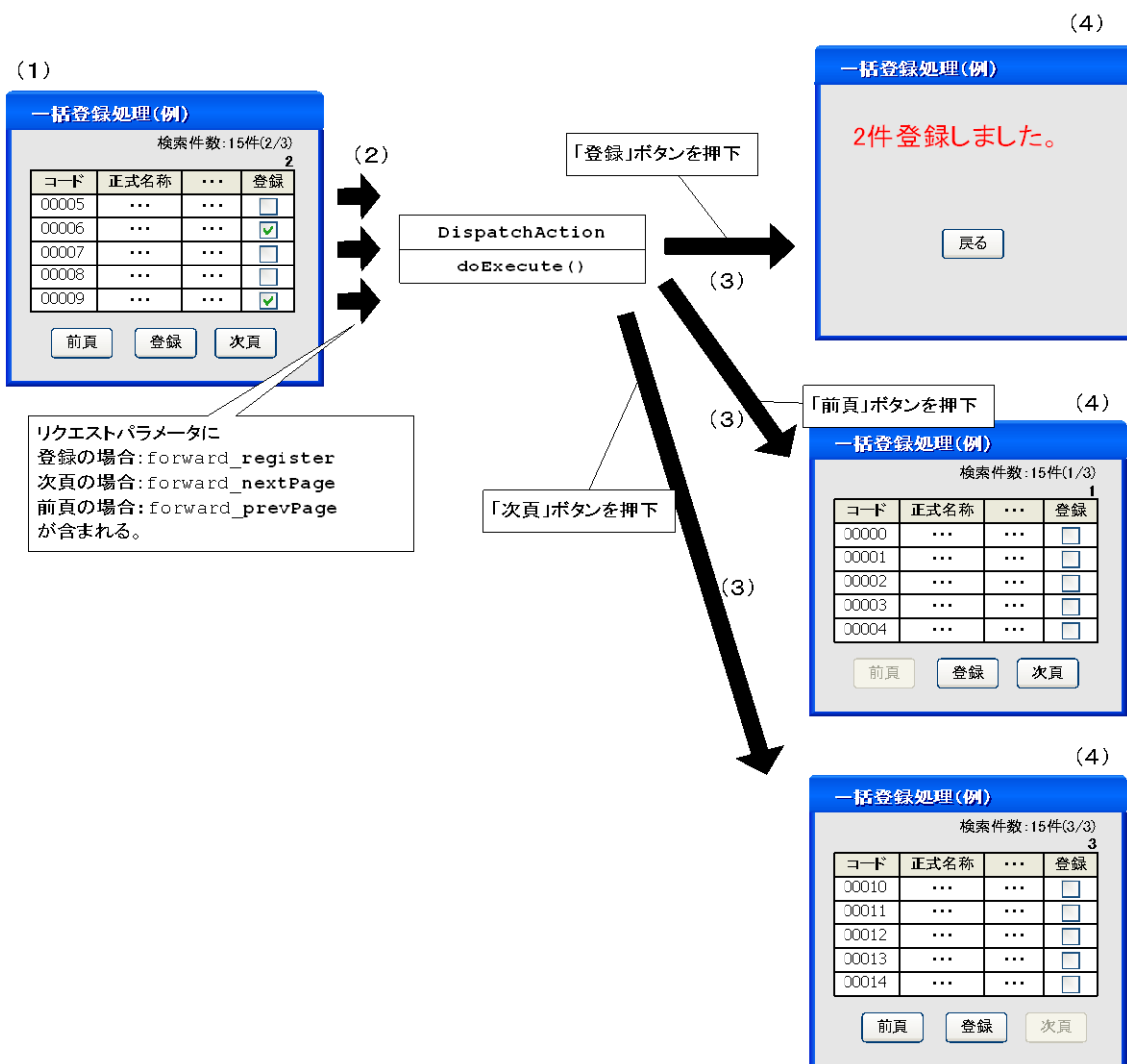
◆ 解説

- (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。
省略した場合、空白画面が表示されるので注意。

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="/fileUploadConfirmDSP" scope="singleton"
      class="jp.terasoluna.fw.web.struts.actions.DispatchAction">
</bean>
```

対応するアクションパス

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>
```

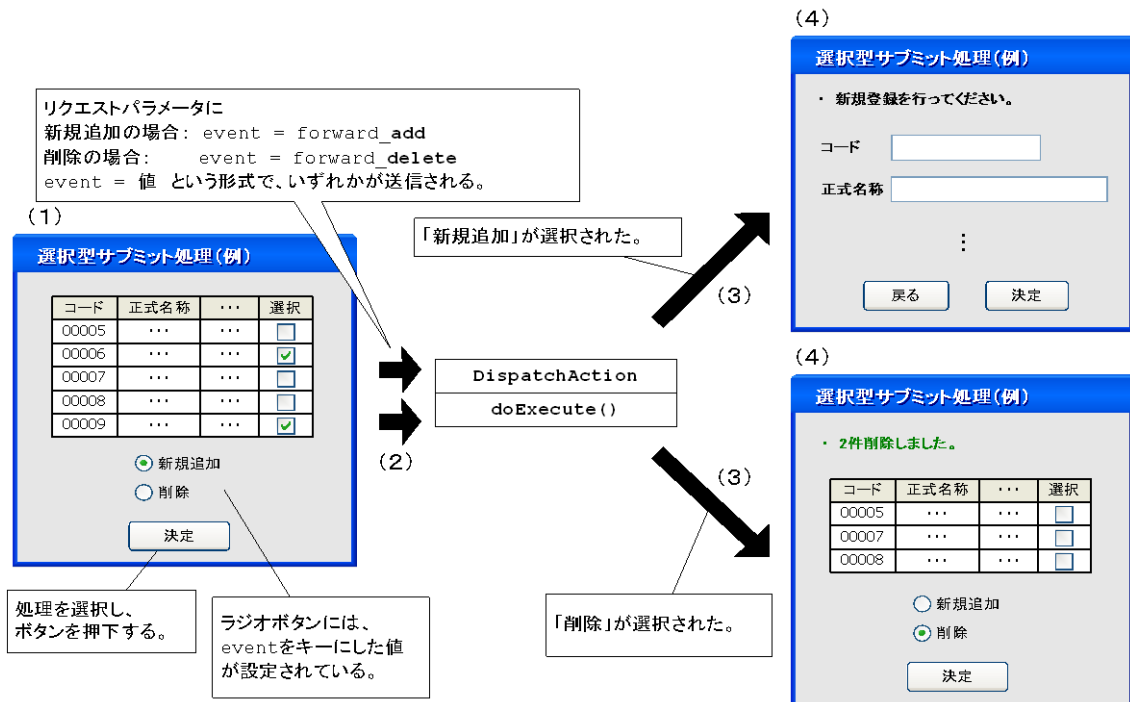
Struts 設定ファイルで定義されたアクションパスを指定する。

“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。
省略した場合、空白画面が表示されるので注意。

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。

● 遷移先の優先順位について

遷移文字列は下記の優先順位で決定される。

1. 上記で指定したリクエストパラメータキーの値で先頭に“forward_”がついているものについて、“forward_”を除去したもの
2. リクエストパラメータキーの先頭に“forward_”が付いているものについて、“forward_”を除去したもの
3. “default”固定（event=“XXXX”、“forward_XXXX”といった 存在し得ない 不正な遷移文字列が指定された場合など）

上記の結果、遷移文字列が“#input”であったとき、“struts-config.xml”の“input”属性が遷移先となる。

“#input”ではないとき、“struts-config.xml”の“forward”要素の内容により遷移先が決定する。

◆ 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。

- **DispatchAction** において **Action** の振り分けをできない場合
ファイルアップロード時にリクエストの対象となる **Action** に **DispatchAction** を設定すると、アップロードしたファイルの大きさが **Struts** に設定した上限ファイルサイズを超えたときに、**DispatchAction** が **Action** の振り分けをできないので注意する必要がある。
Struts の仕様では、ファイルアップロード時に、設定ファイルに記載した上限サイズを超過するリクエストが送信された場合には、リクエストパラメータが空として扱われる。このため振り分け先の **Action** を特定することができず、このような問題が発生する。

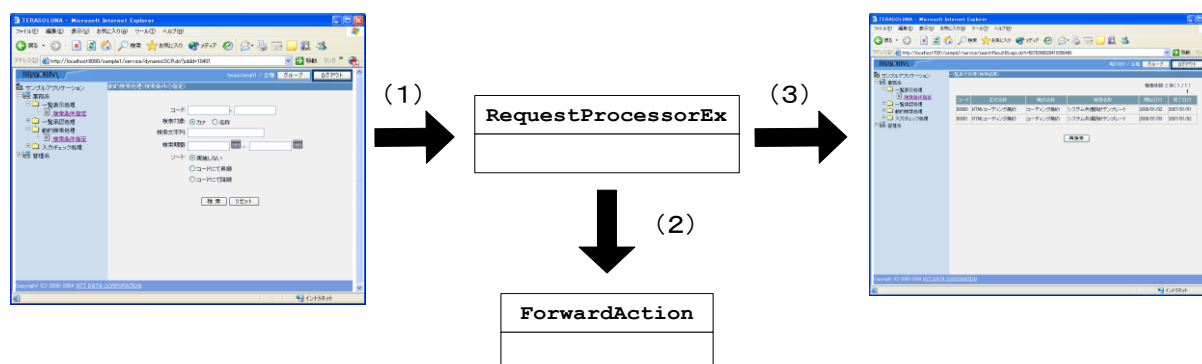
WE-03 フォワード機能

■ 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、以下のようにモジュールからの相対パスを設定する。(contextRelative が false の場合)

➤ Struts 設定ファイル (struts-config.xml)

```
<action path="/foo" parameter="/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

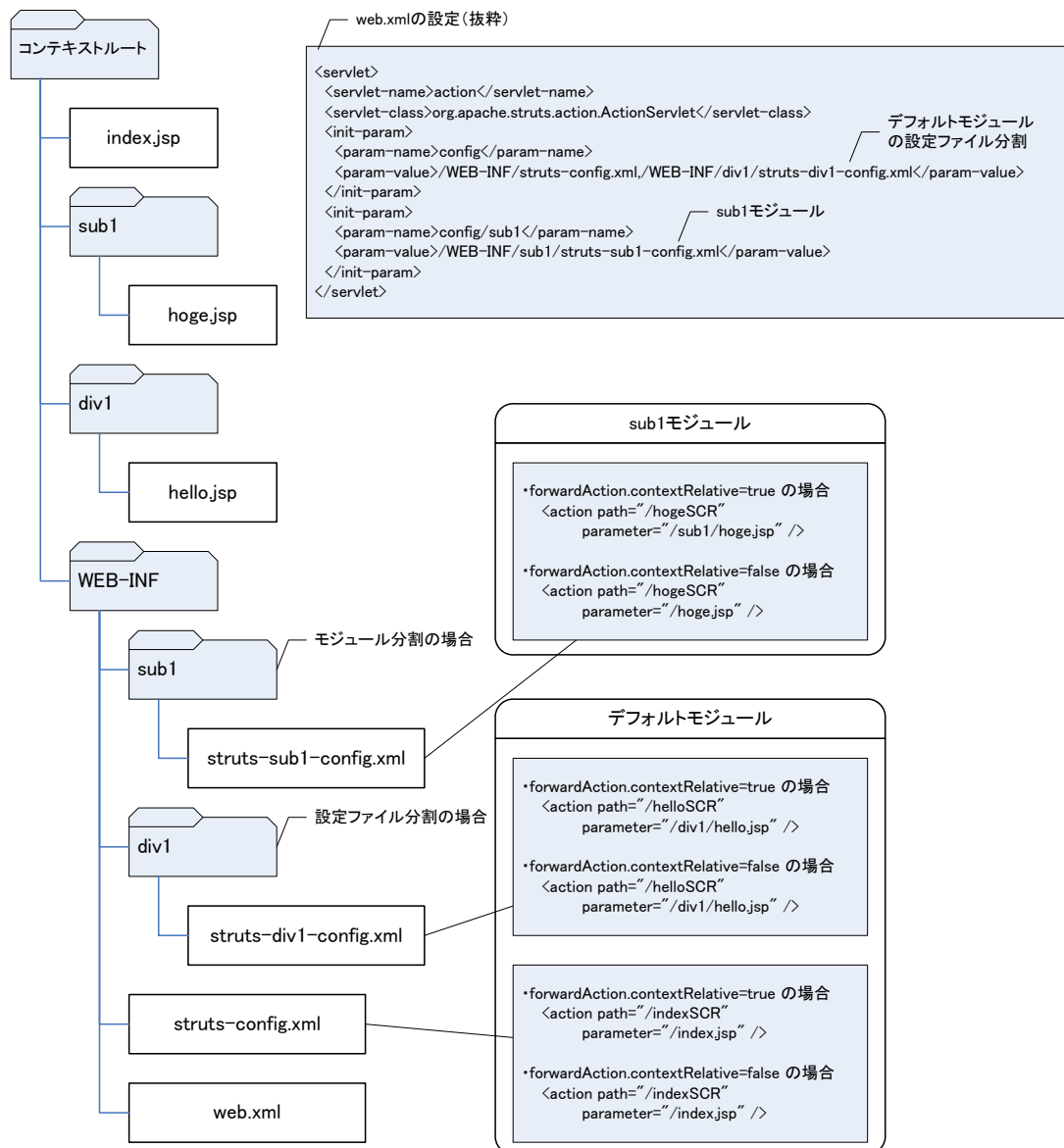
```
<bean name="/foo" scope="prototype"
      class="jp.terasoluna.fw.web.struts.actions.ForwardAction">
</bean>
```

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.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

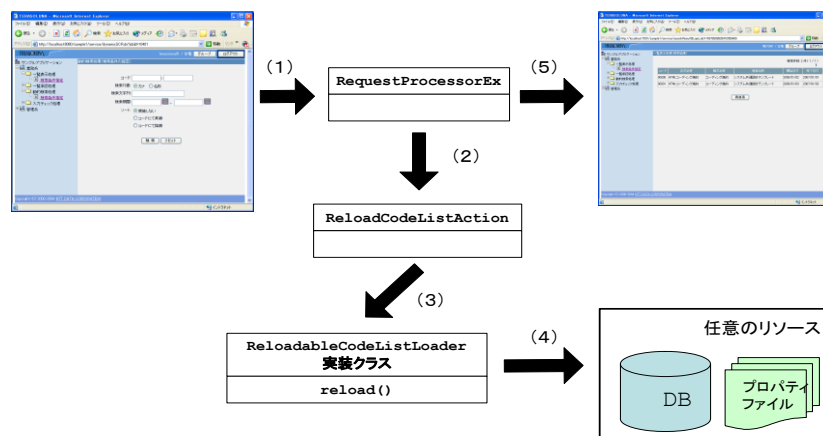
■ 概要

◆ 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再読み込みは不可能である。

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` 設定ファイルのから `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

- コードリスト再読み込み機能の実装は、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">

  <property name="codeListLoader" ref="codeList"/>
</bean>
```

ReloadCodeListAction を指定する。

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="TerasolunaDataSource"/>
  <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

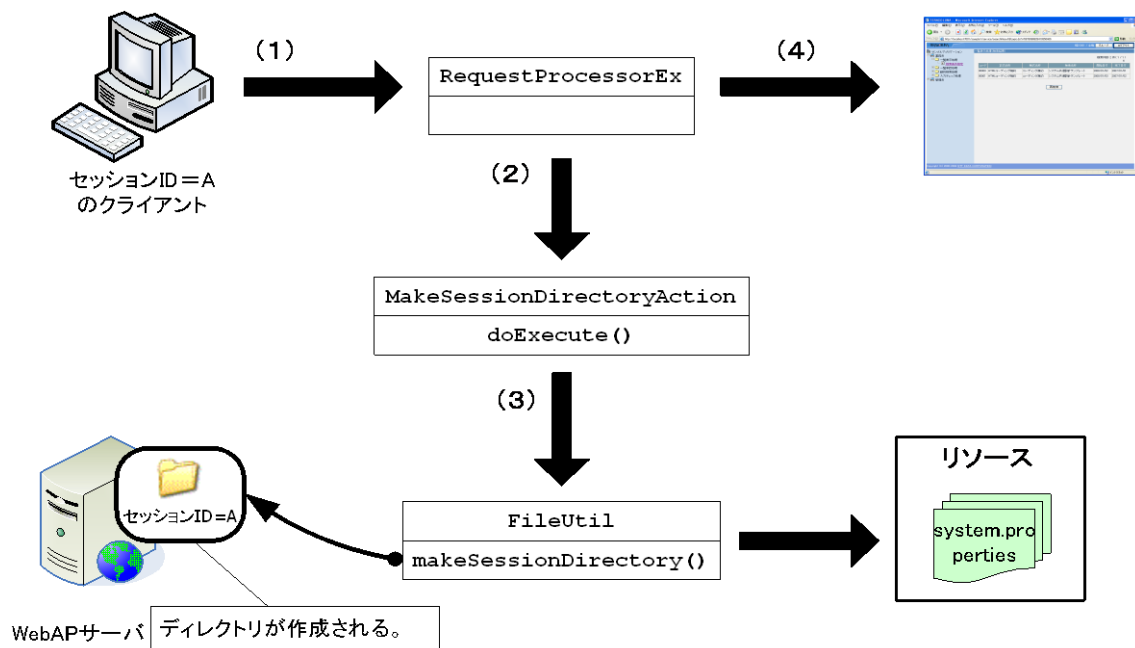
WE-05 セッションディレクトリ作成機能

■ 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

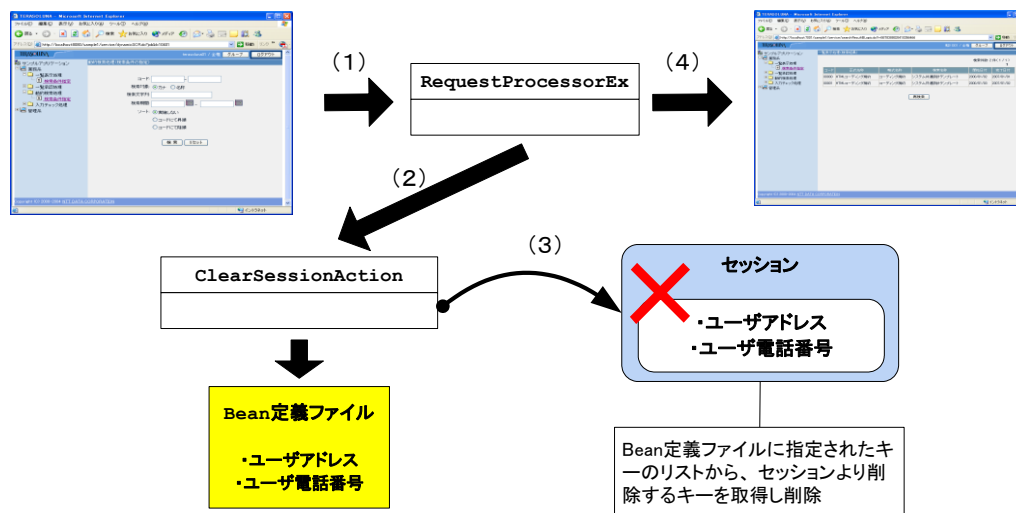
WE-06 セッションクリア機能

■ 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

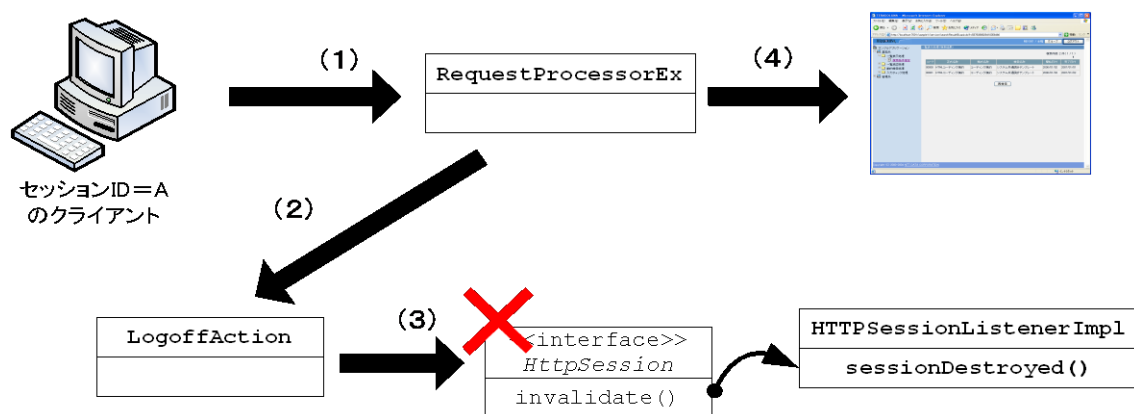
WE-07 ログオフ機能

■ 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。各ルールへ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。」

◆ hankakuKanaString

入力文字列が半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="hankakuKanaString">
    <arg key="label.userName" position="0" />
  </field>
</form>
.....
```

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

半角カナ文字はデフォルトでは以下の文字が該当する。

➤ アイウエオアイウエオカキククサシセソタチツットナニヌネノヒフヘホマミメモヤユョヨラリルロワヅンゝゑゐゐ。」「

半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.hankaku.kana.list=アイウエオアイウエオカキククサシセソタチツットナニヌネノヒフヘホマミメモヤユョヨラリルロワヅンゝゑゐゐ
```

◆ hankakuString

入力文字列が半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="hankakuString">
    <arg key="label.userName" position="0" />
  </field>
</form>
.....
```

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

半角文字は「\ ¢ £ § ¨ ° ± ´ ¶ × ÷」を除く UNICODE の ¥u0000'から ¥u00ff のコードに該当するか、hankakuKanaString ルールに該当するかどうかで判定を行う。hankakuKanaString ルールについては hankakuKanaString ルールの説明を参照のこと。

◆ 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 ウェ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ッン

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} には {1} 文字で入力してください。」

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}には{1}文字で入力してください。」

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「errors.byteRange={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。

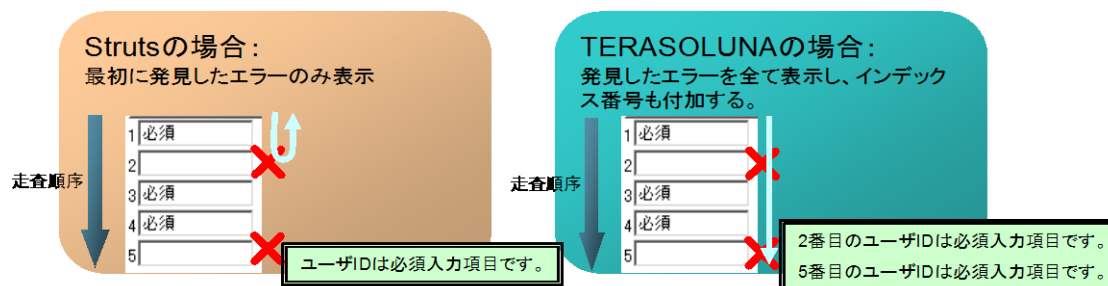
検証結果は boolean 型。エラーの場合は false を返却すること。

- 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。

◆ 注意事項

- 数値範囲（intRange）のチェックをする場合
入力値が数値（int に変換可能）の場合は正常に数値範囲の検証が行われるが、入力値が数値以外（int に変換不可）の場合はフレームワークによって挙動が異なる。TERASOLUNA Server Framework for Java (Rich 版)、TERASOLUNA Batch Framework for Java では、チェックエラーとなる。一方、TERASOLUNA Server Framework for Java (Web 版)では、NumberFormatException が発生し不正終了する。そのため、数値範囲チェックに加えて数値チェックも実施するなどの対応が必要となる。
- 半角スペースのみの文字列をチェックする場合
必須チェック（required）を除く Struts から提供されているルールおよび、TERASOLUNA Server Framework for Java (Web 版)で提供するルールでは、半角スペースのみの文字列が入力値として渡されてきた場合、エラーと判定されない。エラーとする場合は必須チェックと組み合わせるか、半角スペースのチェックを追加すること。

- 日付入力チェックを行う場合の注意点
日付入力チェック (date) では `datePattern` 要素、あるいは `datePatternStrict` 要素に日時のパターンを指定する。しかし本チェックでは指定した日時パターンについて厳密なチェックが行われるわけではない。実際には日付として正しくない値が入力されても、チェックを通ってしまうケースが存在する。これは日付入力チェックの実装で、`SimpleDateFormat` の `parse` メソッドを利用しているためである。
(本事象は Java の既知のバグとしてバグデータベースに登録されている。Bug ID:5055568) 日時のパターンまで厳密なチェックを行う場合には、正規表現チェックを併用すること。
- 数字文字チェック、英数字チェック、大文字英数字チェックの注意点
数字文字チェック (`numericString`)、英数字チェック (`alphaNumericString`)、大文字英数字チェック (`CapAlphaNumericString`) では、行末の改行コード `0x0a` (LF) を検出することはできない。(`0x0d`(CR)や、`0x0d0a`(CRLF)は検出可能)
これにより、文字列の末尾に `0x0a`(LF)を含んだ形でパラメータがわたってきた場合に、入力チェックを通過し、意図しない値がビジネスロジックに渡されるため処理結果に問題が生じる可能性がある。

通常 RFC に準拠したブラウザであれば、改行コードは `0x0d0a`(CRLF)にエンコードされるため問題ないが、悪意を持った攻撃者などがブラウザを介さずに `0x0a`(LF)を含んだリクエストデータを直接送信した場合はこの限りではない。

また、本事象は `FieldChecksEx` の `validateNumericString()` メソッド、`validateAlphaNumericString()` メソッド、`validateCapAlphaNumericString()` メソッドを直接呼び出す場合にも発生する。

厳密なチェックを行う場合には正規表現でチェックを行うか、あるいはビジネスロジックでチェックを行うこと。

正規表現でチェックを行う場合のパターン例を下記に示す。

- `numericString` チェック → `”^[0-9]*(?!¥n)”`
- `alphaNumericString` チェック → `”^[0-9a-zA-Z]*(?!¥n)”`
- `capAlphaNumericString` チェック → `”^[0-9A-Z]*(?!¥n)”`

■ 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

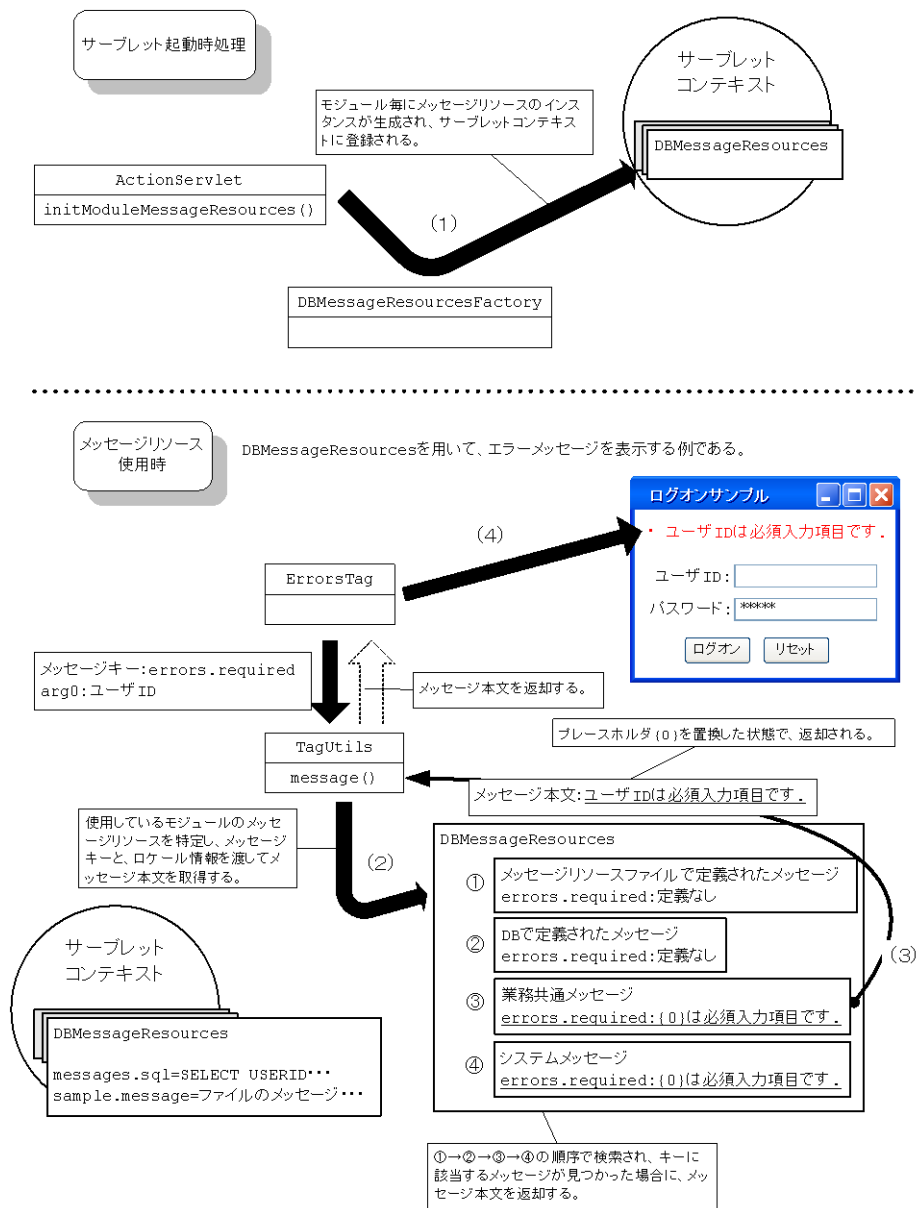
WG-01 メッセージ管理機能

■ 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) サーブレット起動時に、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、サー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。

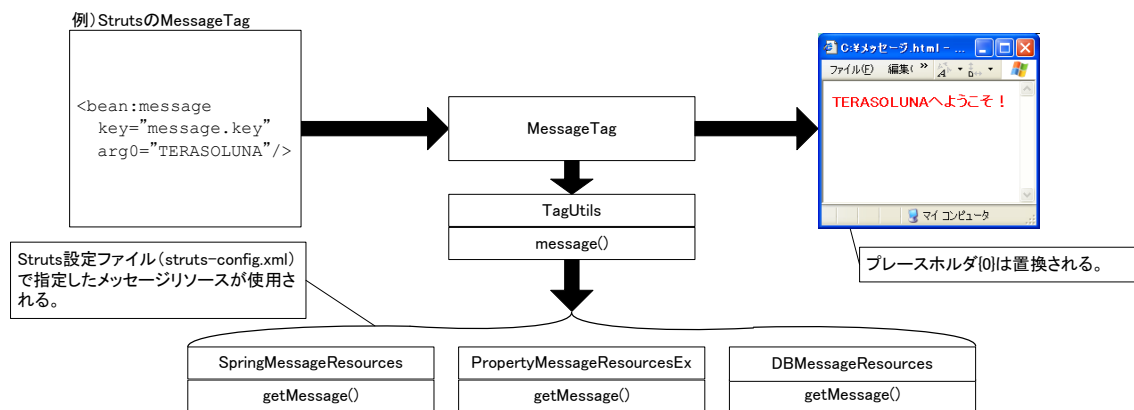
(4) 取得したメッセージを画面に表示する。

■ 使用方法

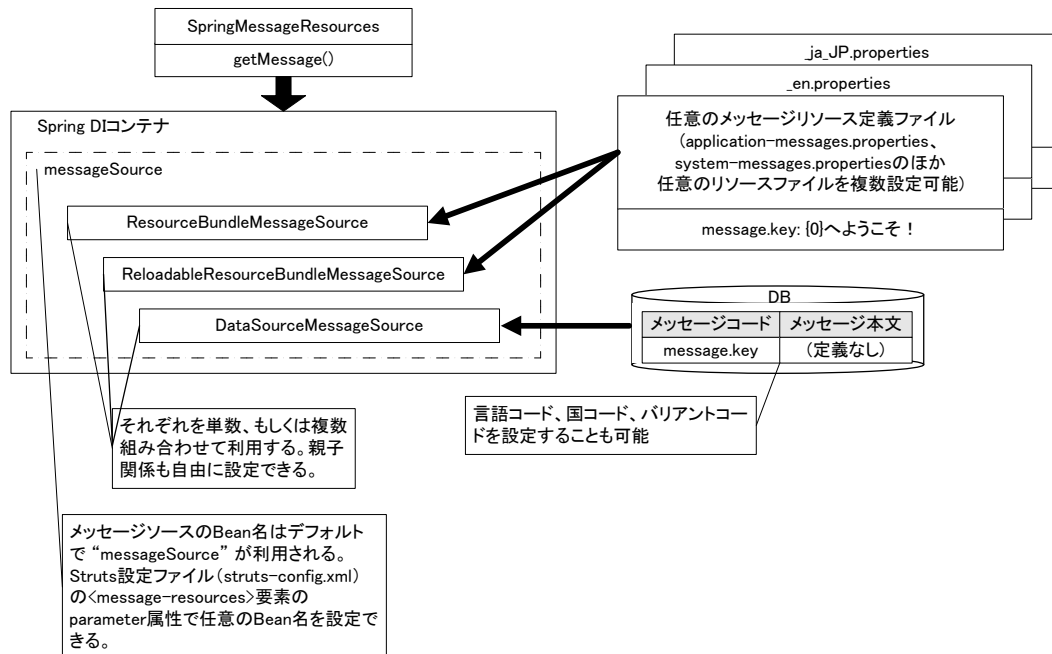
◆ コーディングポイント

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 3 系統がある。

- Spring メッセージリソース
Spring の MessageSource を利用して定義する場合
- 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（Spring メッセージリソースの場合）
MessageSource の設定に準ずる。

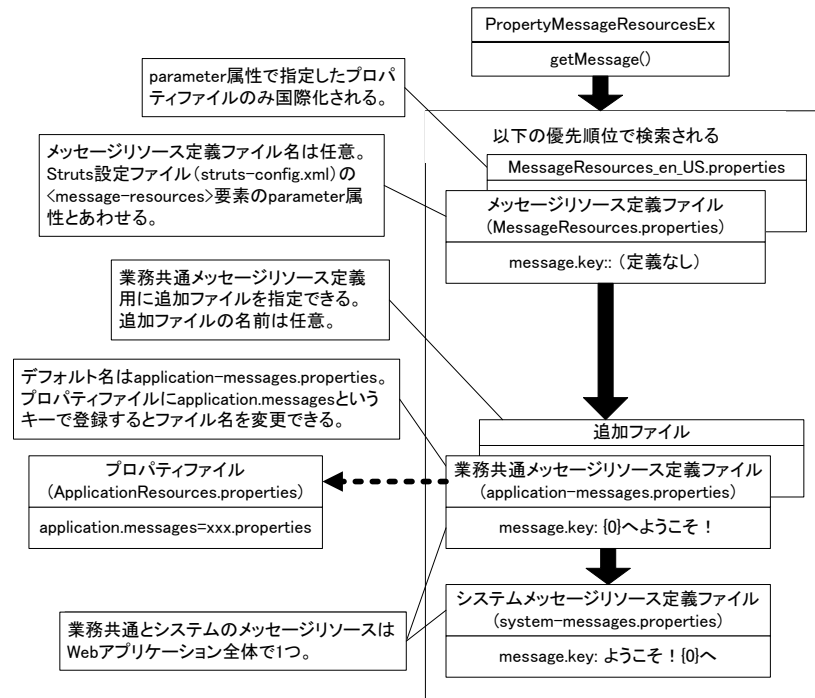


● メッセージリソース定義の優先順位（拡張プロパティメッセージリソースの場合）

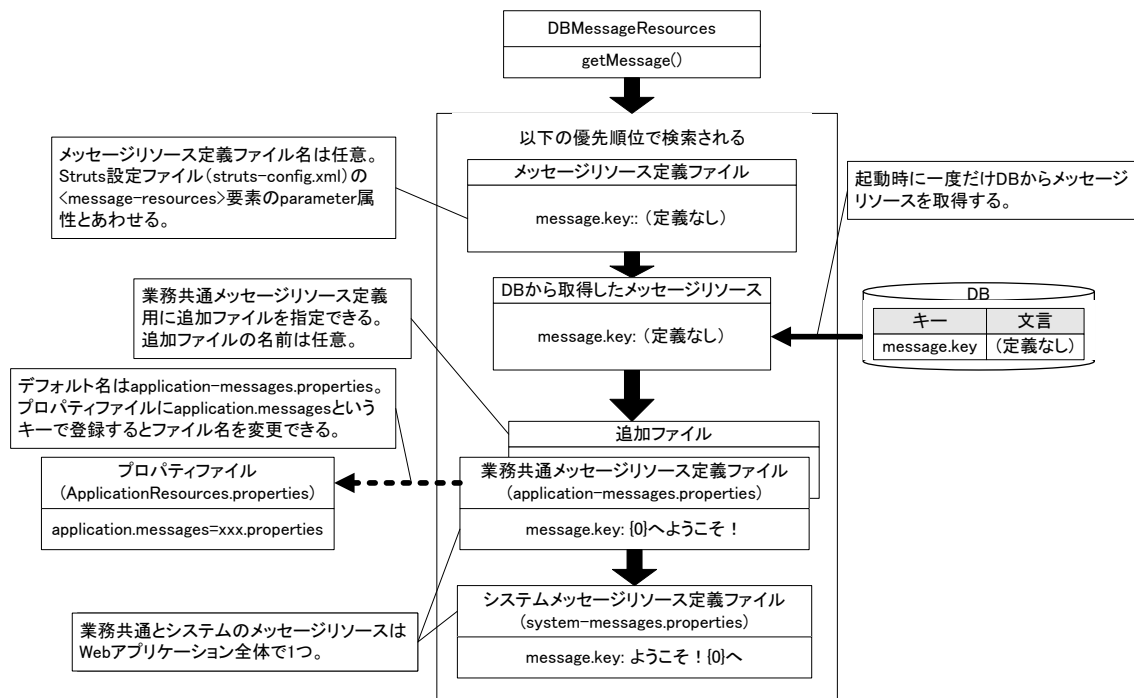
メッセージリソースは以下の場所に定義できる。優先順位順に、

1. Struts 標準のメッセージリソース定義ファイル (MessageResources.properties)
2. 業務共通メッセージリソース定義ファイル (application-messages.properties)
3. 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。



- メッセージリソース定義の優先順位 (DB メッセージリソースの場合)
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. メッセージリソース定義ファイル (MessageResources.properties)
 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.proeptrties** 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
```

メッセージリソースを取得する SQL を指定する。

```
messages.dao=jp.terasoluna.fw.web.struts.action.MessageResourcesDaoImpl
```

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。

業務用件により、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）等と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 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.sql=SELECT MESSAGE_KEY, MESSAGE_VALUE FROM MESSAGES
messages.dao=jp.terasoluna.fw.web.struts.action.MessageResourcesDaoImpl
```

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』
- 『CE-01 メッセージ管理機能』

■ 使用例

- Terasoluna Server Framework for Java (Web 版) 機能網羅サンプル
 - 「UC21 メッセージ管理機能」
 - ◇ /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

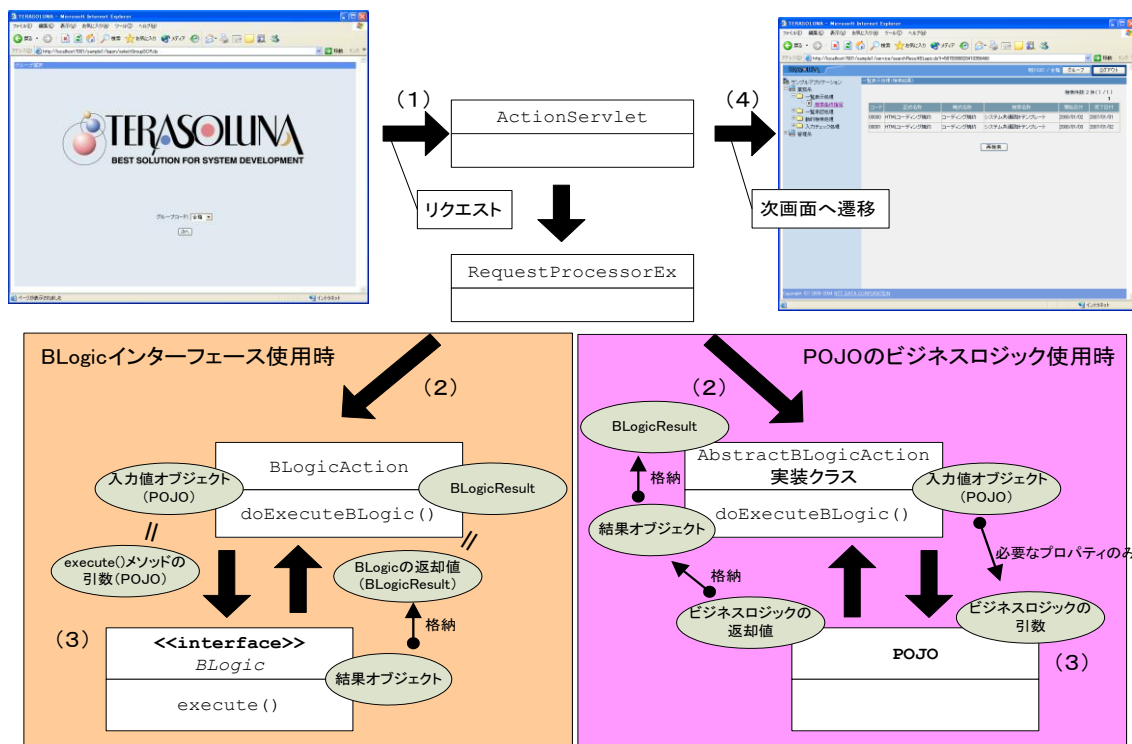
WH-01 ビジネスロジック実行機能

■ 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。

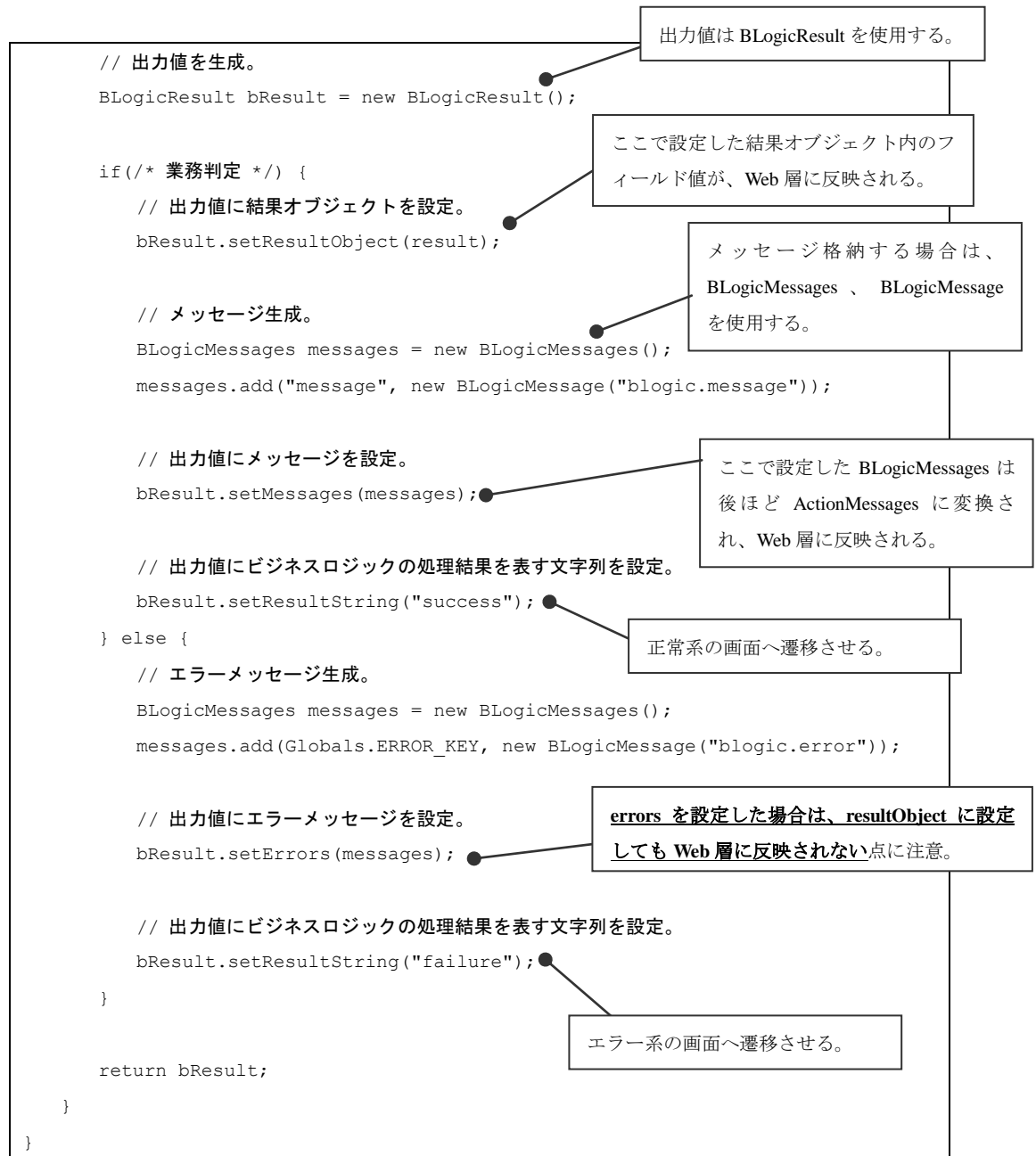
- BLogic インタフェースの実装例

```
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。

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

```
<blogic-params bean-name="jp.terasoluna.xxx.AddBLogicParam">
```

```
<set-property property="fValue1" blogic-property="value1" source="form" />
```

```
<set-property property="fValue2" blogic-property="value2" source="form" />
```

```
</blogic-params>
```

```
<blogic-result>
```

```
<set-property property="sessionValue" blogic-property="result"
```

```
dest="session" />
```

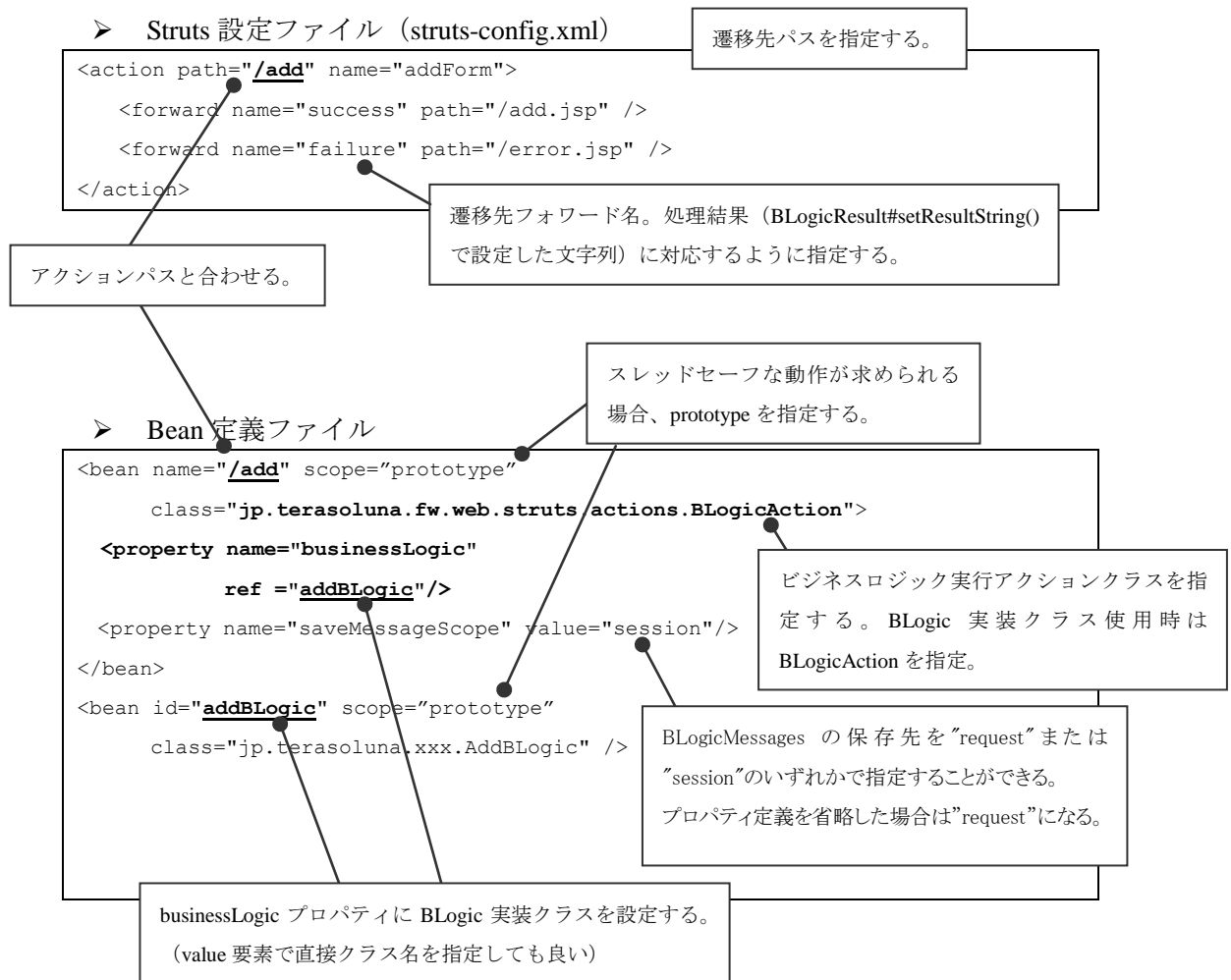
```
</blogic-result>
```

```
</action>
```

入力値 (execute メソッドの引数) を指定する。入力値が存在しない場合は指定しない。

この設定では、フォーム内のフィールド fValue1 の値がビジネスロジック内で AddBLogicParam#getValue1() で取得できるようになる。

この設定では、結果オブジェクト (BLogicResult#setResultObject() で設定したオブジェクト) 内のフィールド result の値が、セッションに "sessionValue" というキーで格納される。
結果オブジェクトの値をプレゼンテーション層に反映する必要がない場合は、blogic-result 要素を省略することができる。



※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="addBLogic"/>
</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

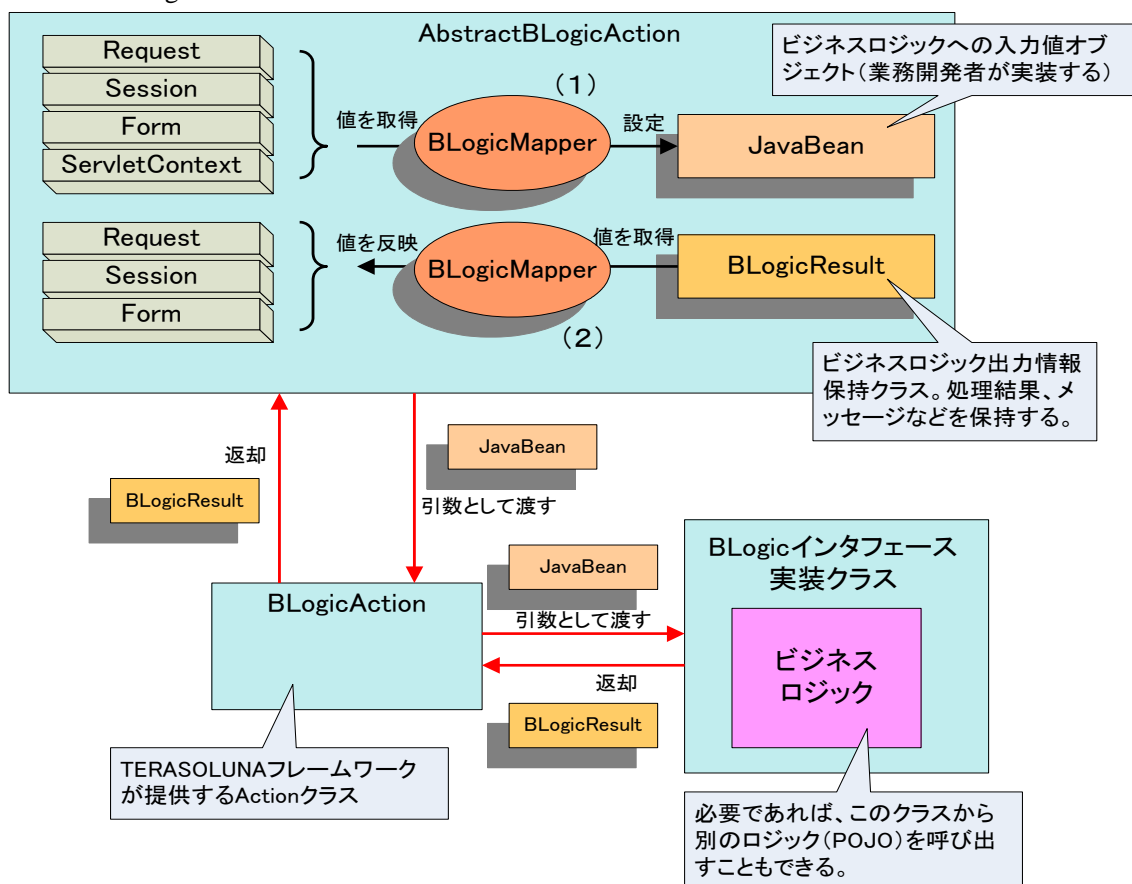
■ 概要

◆ 機能概要

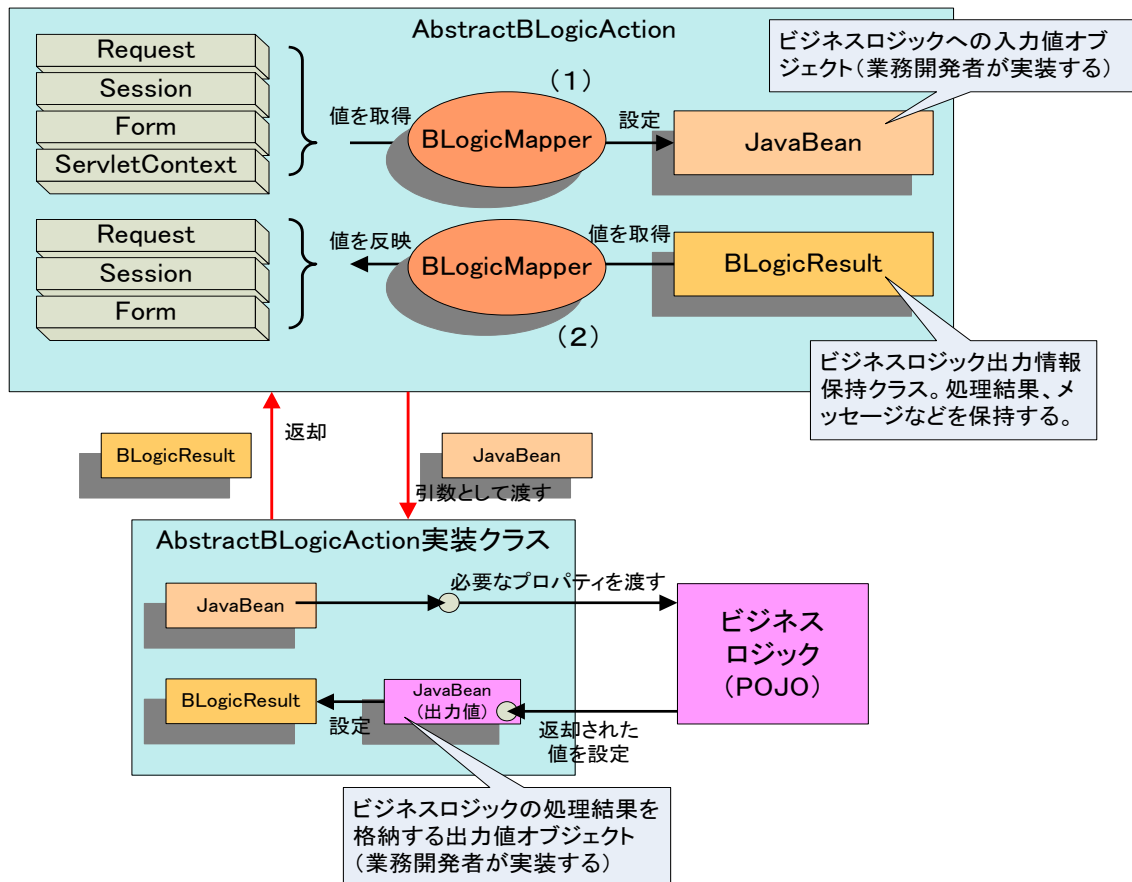
- 設定ファイルに入出力の設定を記述することで、プレゼンテーション層⇄サービス層間のデータの変換を自動化する。
- ビジネスロジック入出力設定の初期化には、Struts のプラグインの機能を利用する。
- デフォルトでは、リクエスト、セッション、アクションフォーム、サーブレットコンテキストからビジネスロジックへの入力処理、ビジネスロジックからリクエスト、セッション、アクションフォームへの出力処理を行える。

◆ 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で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。)

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

※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。

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。
「request」指定ではリクエスト属性(getAttribute)の値のみを取得する。
リクエストパラメータ(getParameter)の値は取得できない点に注意。

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。

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。

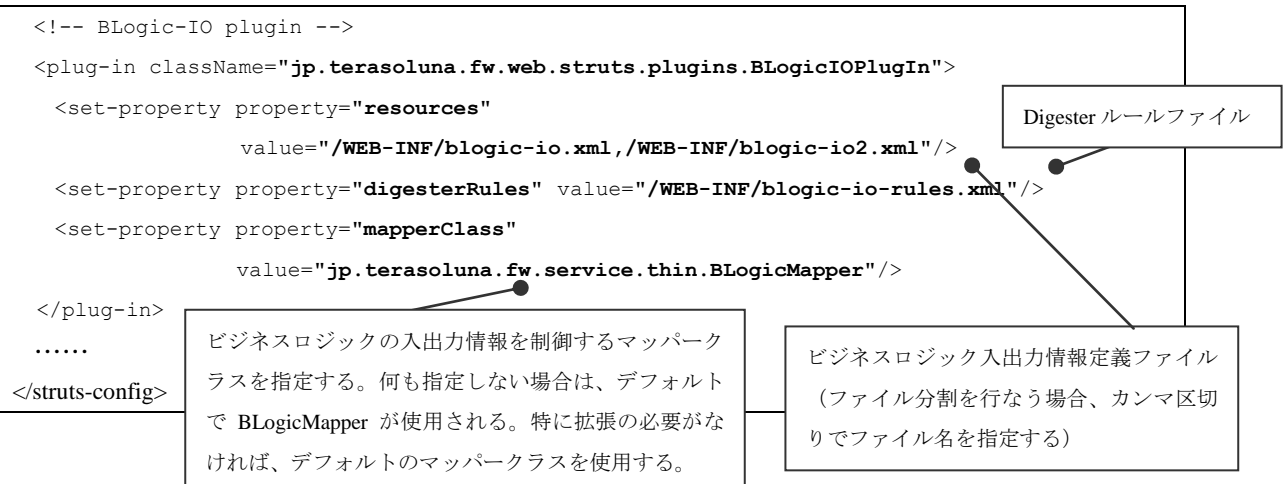
- **Struts 設定ファイル (struts-config.xml)**
 - **pulug-in** 要素として **BLogicIOPlugin** を指定することで、ビジネスロジック入出力機能を使用できる。

```
<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>
```

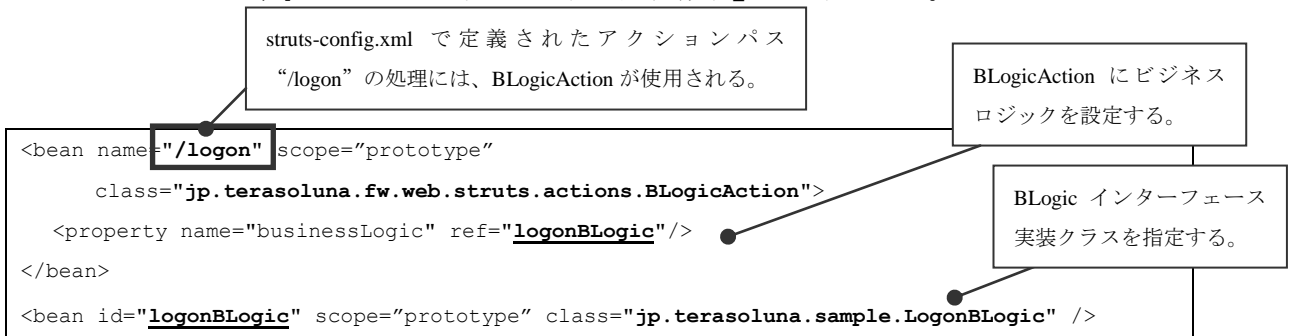
form-bean 要素にて定義したアクションフ
ォーム名を指定する。

(続く)



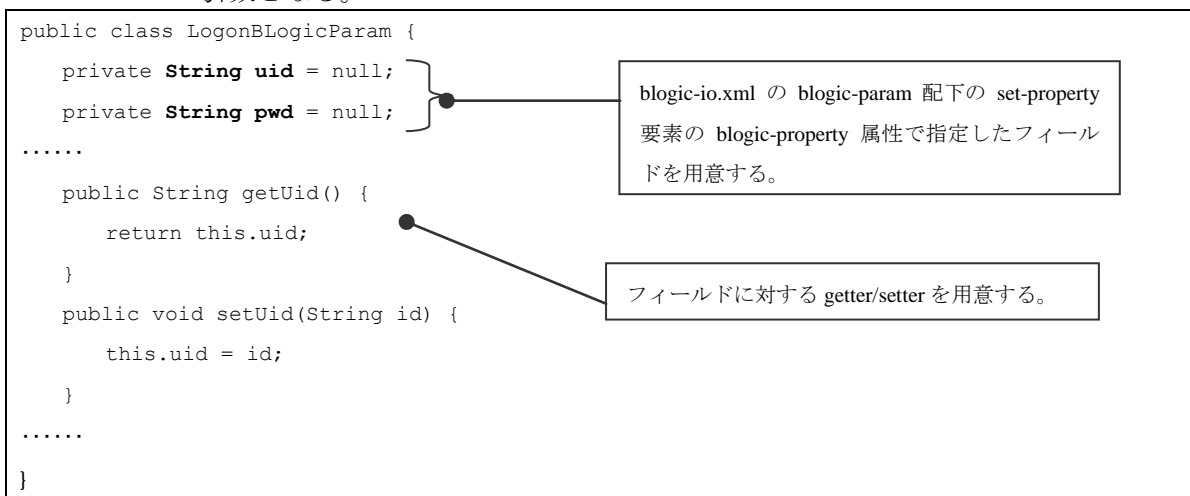
- Bean 定義ファイル

- 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で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 bResult;
}

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。
.....
}
}
```

blogic-io.xml の set-property 要素の blogic-property 属性で指定した名前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、/login.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。詳細な使用方法是 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="listService"/>
</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 stratIndex = 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">
</action>

```

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>

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

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。

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 項目として表示ページのインデックス値がセットされた状態で出力される。また本機能とリセット機能の指定範囲リセットを組み合わせる場合、indexProperty 属性には必ず「startIndex」以外のパラメータ名を指定すること。（indexProperty 属性に「startIndex」と指定してしまうと、リセット機能の指定範囲リセットで使用するパラメータと重複してしまうため）

- JSP

```
<ts:pageLinks action="/list" rowProperty="row"
              totalProperty="totalCount"
              indexProperty="pageStartIndex"
              resetIndex="true"/>
```

一覧表示機能で使用する項目が hidden で出力される

- HTML の出力結果

```
<input type="hidden" name="row" value="10"/>
<input type="hidden" name="pageStartIndex" value="0"/>
<input type="hidden" name="startIndex" value="0"/>
<input type="hidden" name="endIndex" value="9"/>
```

指定範囲リセット機能で使用する項目も hidden で出力される

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 (pageLinks.properties) 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
pageLinks.prev<数値>.char	-	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。
pageLinks.next<数値>.char	-	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。
pageLinks.maxDirectLinkCount	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「 <u>3</u> 4 5 6 <u>7</u> 」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示機能	<t:writeCodeValue>	指定したコードリストの値を画面に表示する。
-------	---------------	--------------------	-----------------------

● 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

現在のログイ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>`
子要素である複数の`<t:ifAuthorized>`タグの権限を組み合わせ、タグ内部の表示／非表示を切り替える。詳細は後述の使用方法例を参照のこと。
`<t:ifAuthorized>`タグを複数使用してアクセス制御を行う場合は以下になる。
前提として、親の`<t:ifAuthroizedBlock>`タグの `blockId` 属性の値を、紐付けたい`<t:ifAuthorized>`タグの `blockId` 属性に対して付与しておく必要がある。
 - `<t:ifAuthorized>`タグの path 属性で指定されたパスに対して1つでもアクセス権がある場合、`<t:ifAuthorizedBlock>`タグの内側の評価を行う。
 - `<t:ifAuthorized>`タグの path 属性で指定されたパスに対して1つもアクセス権がない場合、`<t:ifAuthorizedBlock>`タグの内側の評価は行わない。`<t:ifAuthorizedBlock>`を更に入れ子にする場合は、親要素の `blockId` 属性と子要素の `parentBlockId` 属性を同一にすることで、表示制御を紐付けることができる。

◆ 使用方法例

● <t:ifAuthorized>タグ、<t:ifAuthorizedBlock>タグの使用例

メニュー画面にて、アクセス権毎でメニュー内容の表示/非表示切り替えを行う。
アクセス権限は以下の通りとする。

- ① 管理者権限 ⇒ 全てのパスにアクセス権あり。
- ② 一般A権限 ⇒ "/sample/search.do"にのみアクセス権あり。
- ③ 一般B権限 ⇒ 全てのパスにアクセス権なし。

sampleMain.jsp

```

.....
<t:ifAuthorizedBlock blockId="ABC" >■サンプル ユーザ情報
  <t:ifAuthorizedBlock blockId="EFG" parentBlockId="ABC" >
    ※管理者権限と一般A権限のユーザのみ表示されるメニューです。
    <t:ifAuthorized path="/sample/insert.do" blockId="EFG" >
      <html:link href="sample/insert.do">サンプル機能 新規登録</html:link>
    </t:ifAuthorized>
    <t:ifAuthorized path="/sample/edit.do" blockId="EFG" >
      <html:link href="sample/edit.do">サンプル機能 更新・削除</html:link>
    </t:ifAuthorized>
    <t:ifAuthorized path="/sample/search.do" blockId="EFG" >
      <html:link href="sample/search.do">サンプル機能 参照</html:link>
    </t:ifAuthorized>
  </t:ifAuthorizedBlock>
</t:ifAuthorizedBlock>
.....

```

<t:ifAuthorizedBlock>タグによる権限チェック領域
ボディ内の parentBlockId="ABC"で紐付けられた領域が権限を持つ場合にのみ表示される。
※ ①,②の場合に表示され、③の場合は何も表示されない。

<t:ifAuthorizedBlock>タグによる権限チェック領域
ボディ内の blockId="EFG"で紐付けられた領域が権限を持つ場合にのみ表示される。
※ ①,②の場合に表示され、③の場合は何も表示されない。

<t:ifAuthorized>タグによる権限チェック領域
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表示される。
外側の<t:ifAuthorizedBlock>タグに対し、blockId を紐付ける。
※ ①の場合、機能リンクが全て表示される。
②の場合、『サンプル機能 参照』リンクのみが表示される。
③の場合、何も表示されない。

↓上記 JSP 出力結果

- ① 管理者権限 ⇒ 全ての機能が表示される。

■サンプル ユーザ情報
※管理者権限と一般A権限のユーザのみ表示されるメニューです。

サンプル機能 新規登録
サンプル機能 更新・削除
サンプル機能 参照

- ② 一般A権限 ⇒ 『サンプル機能 参照』のみ表示される。

■サンプル ユーザ情報
※管理者権限と一般A権限のユーザのみ表示されるメニューです。

サンプル機能 参照

- ③ 一般B権限 ⇒ 何も表示されない。

※アクセス権チェックの実装方法・例については、『WA-02 アクセス権限チェック機能』を参照のこと。

◆ 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<t: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<t: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

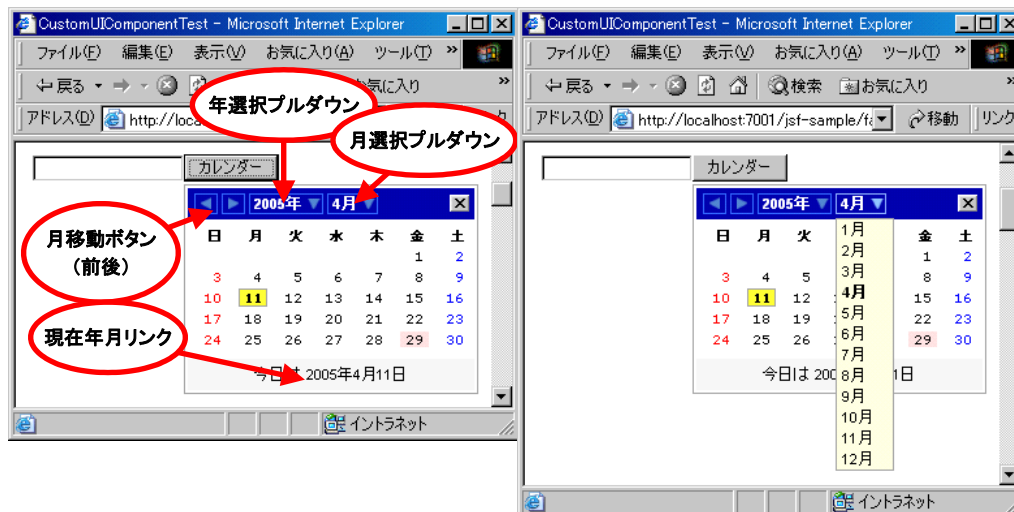
◆ 概要

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。

◆ 注意点

- “event”という名の要素は利用不可

本機能を利用する画面で“event”という名で要素を定義すると本機能が誤動作する。この場合は“event”という要素は利用しないこと。

(JavaScript の event オブジェクトを利用している為)

特に WE-02 標準ディスパッチャ機能 を利用して遷移先条件をラジオボタンなどで設定する場合は、デフォルトでは遷移先を“event”という要素に設定するため注意が必要である。この場合は DispatchAction の Bean 定義にて、“event”プロパティを設定し、デフォルトの “event” から別の項目名に変更しておくこと。

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

この場合 “dispatchName” という名でラジオボタンを定義する。

◆ 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、 &nbsp; 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、 &nbsp; 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、format 属性で指定した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。
format	-	value 属性で日付を指定する際の日付時刻のフォーマット。デフォルト値は “yyyy/MM/dd HH:mm:ss”

◆ 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、format 属性で指定した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。
format	-	value 属性で日付を指定する際の日付時刻のフォーマット。デフォルト値は“yyyy/MM/dd HH:mm:ss”

◆ 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

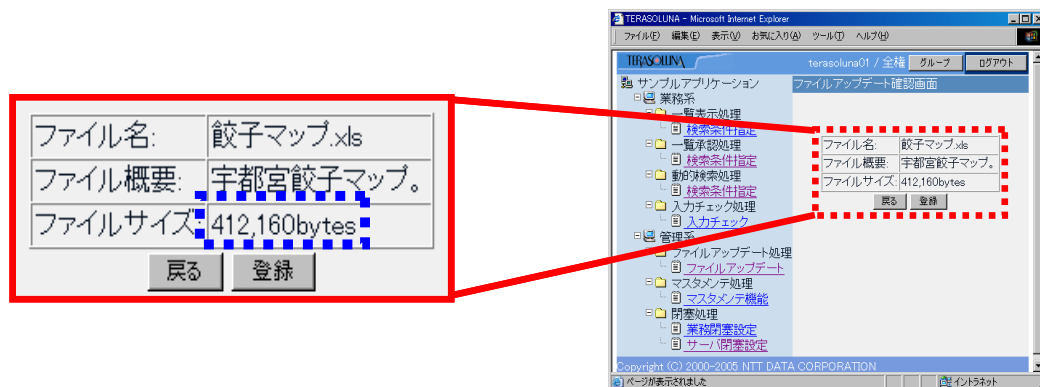
◆ 概要

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:trim>、<t:ltrim>、<t:rtrim>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。

◆ 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

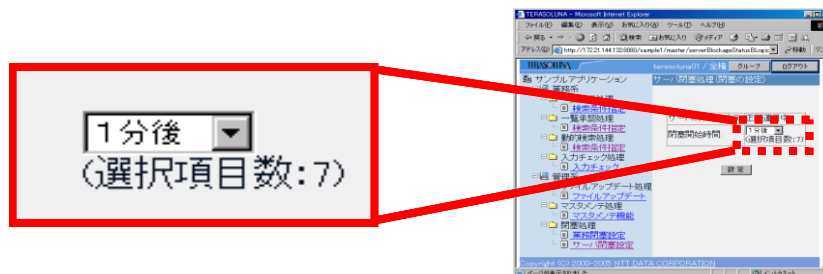
読み込み済みのコードリストを、jsp 内で利用する。

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

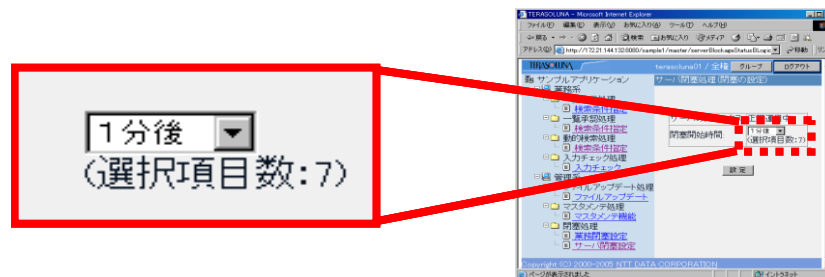
読み込み済みのコードリストの要素数を出力する。

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.*

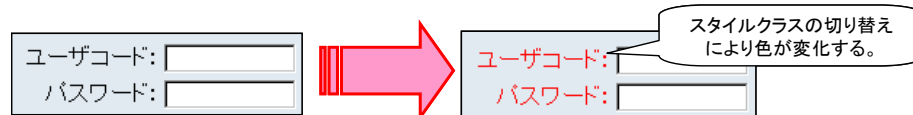
■ WK-01 スタイルクラス切り替え機能

◆ 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.*

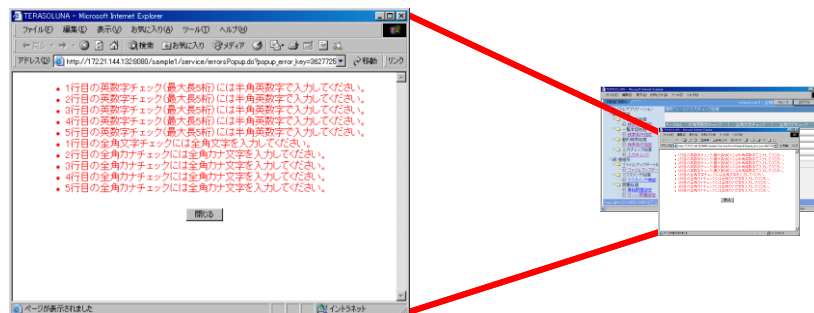
■ WK-02 メッセージ表示機能

◆ 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 を拡張していて、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 を拡張していて、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。



◆ 機能詳細

- メッセージリソースファイルの登録
詳細は『CE-01 メッセージ管理機能』『WG-01 メッセージ管理機能』を参照。
- メッセージリソース定義 (application-message.properties)
表示する文字列は、メッセージキー情報を元にメッセージリソースから取得する。

application-message.properties 記述例

```
errors.alphaNumericString={0}には半角英数字で入力してください。  
errors.hankakuKanaString={0}には半角カナ文字を入力してください。  
errors.hankakuString={0}には半角文字を入力してください。  
errors.zenkakuString={0}には全角文字を入力してください。
```

メッセージキー情報と対応する文字列
を対で定義する。

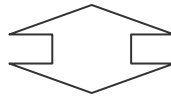
- 文字出力時のレイアウト
メッセージリソース定義内で以下の属性を指定することで、様々なレイアウトでの文字出力が可能となる。

【属性】

- errors.header = メッセージ全体の前に表示される。
- errors.footer = メッセージ全体の後に表示される。
- errors.prefix = 各メッセージの前に表示される
- errors.suffix = 各メッセージの後に表示される

application-message.properties 記述例

```
errors.header=<div style="color:red">
errors.footer=</div>
errors.prefix=>
errors.suffix=<br/>
password.short=パスワードは5文字以上で入力して下さい。
username.required=ユーザ名は必須です。
```



メッセージ出力結果例

```
<div style="color:red">
>ユーザ名は必須項目です。<br/>
>パスワードは5文字以上を入力して下さい。<br/>
</div>
```

- メッセージの設定方法（アクションクラス）
メッセージの設定は、BLogicMessages クラスの add メソッドで各メッセージを格納します。各メッセージは BLogicMessage クラスを使用して、メッセージリソースからメッセージを取得します。
<ts:errors>タグでエラー情報を表示させたい場合は、BLogicResult の setErrors メソッドで、エラーメッセージを格納します。
<ts:messages>タグでメッセージ情報を表示させたい場合は、BLogicResult の setMessages メソッドでメッセージを格納します。

エラーメッセージ設定例

```
// エラー情報生成
BLogicMessages errors = new BLogicMessages();
errors.add(Globals.ERROR_KEY, new BLogicMessage("errors.required","ユーザID");

//エラー情報を保存する。
BLogicResult result = new BLogicResult();
result.setErrors(errors);
```

◆ 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、サブミットボタンごとに<form>タグのターゲットを指定できる。

◆ 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。指定できる値は以下の通り。 <ul style="list-style-type: none">● _blank 新たにブラウザを立ち上げて表示● _top フレームの分割を廃止して画面全体で表示● _self リンク元と同じウィンドウ(フレーム)に表示● _parent 親フレームに表示● ウィンドウ名(フレーム名) 任意のウィンドウ(フレーム)に表示

➤ 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.*

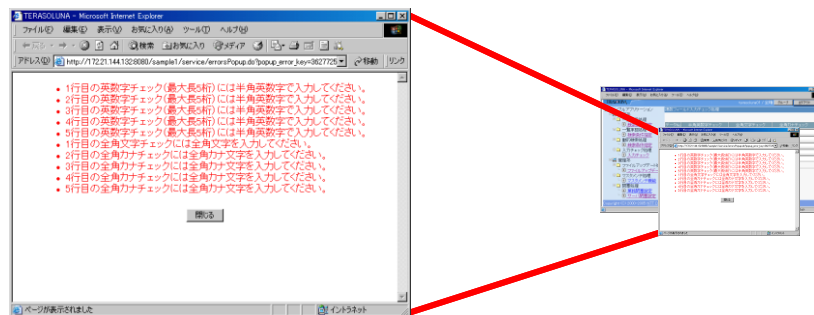
■ WK-06 メッセージポップアップ機能

◆ 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。詳細な使用方法是 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

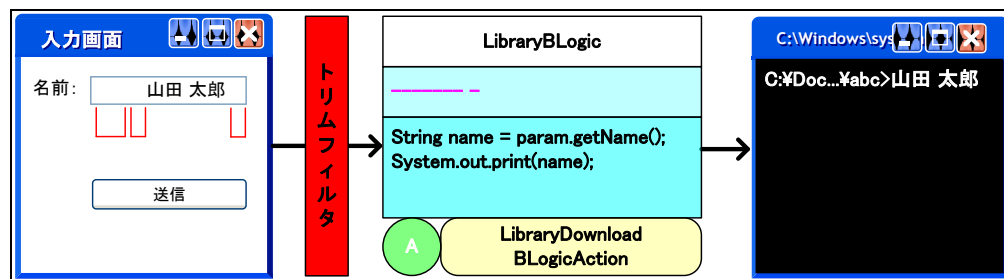
AL008-01 リクエストパラメータトリム機能

■ 概要

◆ 機能概要

- クライアントが送信した入力値を自動的にトリミングするサーブレットフィルタを提供する。
- 以下、このサーブレットフィルタを「トリムフィルタ」と呼ぶ。

◆ 概念図



◆ 解説

- クライアントが送信した入力値を自動的にトリミングする。
- ビジネスロジックには、トリミングされた状態で入力値が受け渡される。そのため、個別のビジネスロジックで改めてトリミングを行う必要はない。
- フォームデータ送信時のエンコード形式に「multipart/form-data」が指定されている場合、トリミングは行われず。この場合、必要に応じて各自がトリミングを行うこと。

■ 使用方法

◆ コーディングポイント

- サーブレットフィルタの設定
クライアントの入力値を自動的にトリミングしたい場合、Web アプリケーション設定ファイル（web.xml）で専用のサーブレットフィルタ（TrimRequestParamFilter）を登録する。その際、初期化パラメータとして、リクエストパラメータのトリムに使用する正規表現パターン（trimPattern）を指定する。以下は、リクエストパラメータ前後の前後の全角・半角スペースを自動的に除去したい場合の trimPattern の指定例である。

1. Web アプリケーション設定ファイル (/WEB-INF/web.xml)

```
<filter>
  <filter-name>trimRequestParameterFilter</filter-name>
  <filter-class>jp.terasoluna.fw.ex.web.thin.TrimRequestParameterFilter</filter-class>
  <init-param>
    <param-name>trimPattern</param-name>
    <param-value>^[ \t\S]*[ \t\S]*$</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name> trimRequestParameterFilter </filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw. ex.web.thin.ParameterTrimmedHttpServletRequest	指定された正規表現パターンに基づいてトリムされたリクエストパラメータを返す HttpServletRequest の実装クラス。
2	jp.terasoluna.fw. ex.web.thin. TrimRequestParameterFilter	指定された正規表現パターンに基づいて、クライアントからのリクエストパラメータをトリムするクラス。

■ 備考

Web アプリケーション設定ファイルに記述した trimPattern が正規表現パターンとして正しくない場合、warn レベルのログを出力し、トリムは行われません。

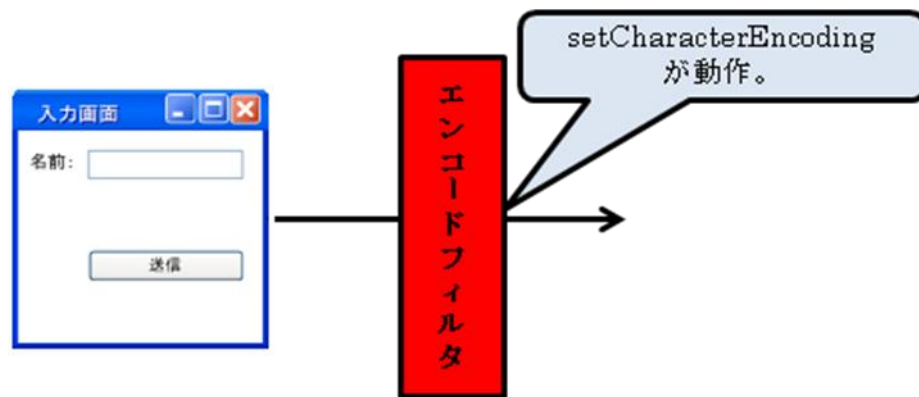
AL008-02 エンコードフィルタ機能

■ 概要

◆ 機能概要

- クライアントが送信リクエストの文字エンコードを設定できるサーブレットフィルタを提供する。
- 以下、このサーブレットフィルタを「エンコードフィルタ」と呼ぶ。

◆ 概念図



◆ 解説

- クライアントが送信したリクエストに対し、`setCharacterEncoding` が実行される。
- フィルタで文字エンコードが実行されるため、jsp・ビジネスロジックでは文字エンコードを行う必要はない。

■ 使用方法

◆ コーディングポイント

- サーブレットフィルタの設定
クライアントのリクエストに自動的に文字エンコード指定を行いたい場合、Webアプリケーション設定ファイル（`web.xml`）で専用のサーブレットフィルタ（`SetCharacterEncodingFilter`）を登録する。その際、初期化パラメータとして、リクエストの文字エンコード種別・フィルタの複数回処理フラグ・フィルタ無効化フラグを指定する。以下は、リクエストに自動的に文字エンコード指定を行う場合の `encoding` の指定例である。

1. Web アプリケーション設定ファイル (/WEB-INF/web.xml)

```
<filter>
  <filter-name>encodingFilter</filter-name>
  <filter-class>
    jp.terasoluna.fw.ex.web.thin.SetCharacterEncodingFilter
  </filter-class>
  <init-param>
    <param-name>encoding</param-name>
    <param-value>Windows-31J</param-value>
  </init-param>
  <init-param>
    <param-name>more</param-name>
    <param-value>>false</param-value>
  </init-param>
  <init-param>
    <param-name>ignore</param-name>
    <param-value>>false</param-value>
  </init-param>
</filter>

<filter-mapping>
  <filter-name>encodingFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw. ex.web.thin. SetCharacterEncodingFilter	リクエストパラメータのエンコーディング設定を行うクラス。

■ 備考

Web アプリケーション設定ファイルに記述した `encoding` がエンコード名として正しくない場合や、使用できないエンコードの場合、エンコーディングは行われな

い。

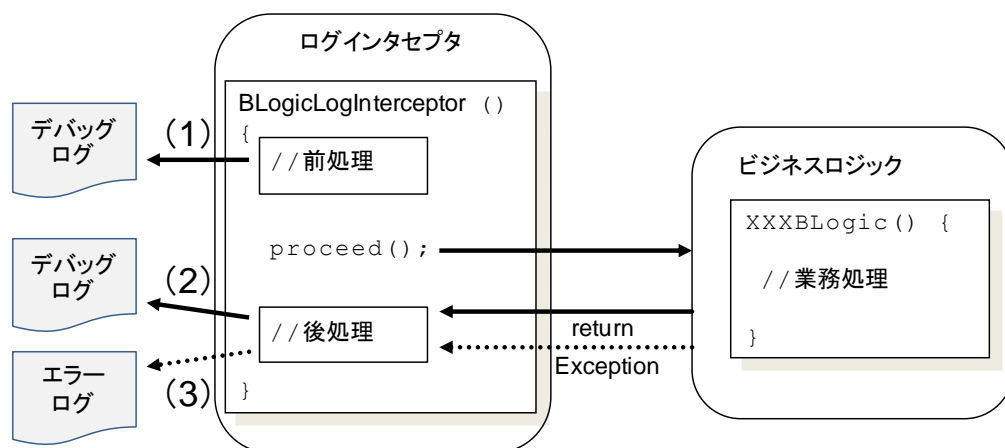
AL009-01 デバッグログ出力機能

■ 概要

◆ 機能概要

- ビジネスロジックの実行前後にデバッグログを出力するインタセプタを提供する。
- Terasoluna Server Framework for Java web 版の BLogic インタフェースを実装したクラスがログ出力の対象となる。
- Spring の AOP 機能を利用してログ出力をする。

◆ 概念図



◆ 解説

- ログインタセプタよりビジネスロジックが実行される前にデバッグログを出力する。
- ビジネスロジック実行後、正常終了の場合、デバッグログが出力される。
- ビジネスロジック実行後、異常終了の場合、エラーログまたはデバッグログが出力される。

■ 使用方法

◆ コーディングポイント

ビジネスロジックの実行前後にデバッグログを出力できる。ビジネスロジック内で例外が発生した場合は、デフォルトの設定ではエラーログが出力される。特定の例外とそのサブクラスの例外に対してエラーログ出力の対象外としたい場合、プロパティ属性に対象となる例外クラスの完全修飾クラス名を記述する。

- Bean 定義ファイルの設定

Bean 定義ファイル（applicationContext.xml）にデバッグログ出力用インタセプタ、ポイントカット、アドバイザの定義を記述する

➤ Bean 定義ファイル（applicationContext.xml）

```
<!-- デバッグログ出力用インタセプタ -->
<bean id="blogicLogInterceptor"
class="jp.terasoluna.fw.ex.aop.log.BLogicLogInterceptor" >
  <property name="noErrorLogExceptionLists">
    <list>
      <value>com.example.XXXException</value>
      <value>…略…</value>
    </list>
  </property>
</bean>
```

エラーログとして出力したくない実行例外の完全修飾クラス名を記述する

```
<!-- ポイントカット、アドバイザの定義-->
<aop:config>
  <aop:pointcut id="blogicBeans" expression="bean(*BLogic)" />
  <aop:advisor pointcut-ref="blogicBeans" advice-ref="blogicLogInterceptor" />
</aop:config>
```

➤ ログファイル出力例（正常系）

```
[2010/12/06 17:02:34][DEBUG][BLogicLogInterceptor]
LogOutputBLogic is being executed...
[2010/12/06 17:02:34][DEBUG][BLogicLogInterceptor] Params:null
[2010/12/06 17:02:34][DEBUG][BLogicLogInterceptor]
Result:resultString=success,resultObject=null
[2010/12/06 17:02:34][DEBUG][BLogicLogInterceptor]
LogOutputBLogic was executed completely.
```

➤ ログファイル出力例（異常系：XXXException 以外の実行例外発生）

```
[2010/12/06 17:05:41][DEBUG][BLogicLogInterceptor]
  LogOutputBLogic is being executed...
[2010/12/06 17:05:41][DEBUG][BLogicLogInterceptor] Params:null
[2010/12/06 17:05:41][ERROR][BLogicLogInterceptor]
com.example.YYYException
…例外スタックトレース…
```

➤ ログファイル出力例（異常系：XXXException の実行例外発生）

```
[2010/12/06 17:05:41][DEBUG][BLogicLogInterceptor]
  LogOutputBLogic is being executed...
[2010/12/06 17:05:41][DEBUG][BLogicLogInterceptor] Params:null
[2010/12/06 17:05:41][DEBUG][BLogicLogInterceptor]
com.example.XXXException
…例外スタックトレース…
```

■ リファレンス

◆ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.ex.aop.log.BLogicLogI nterceptor	BLogic クラスのデバッグログを出力するためのインタセ プタ

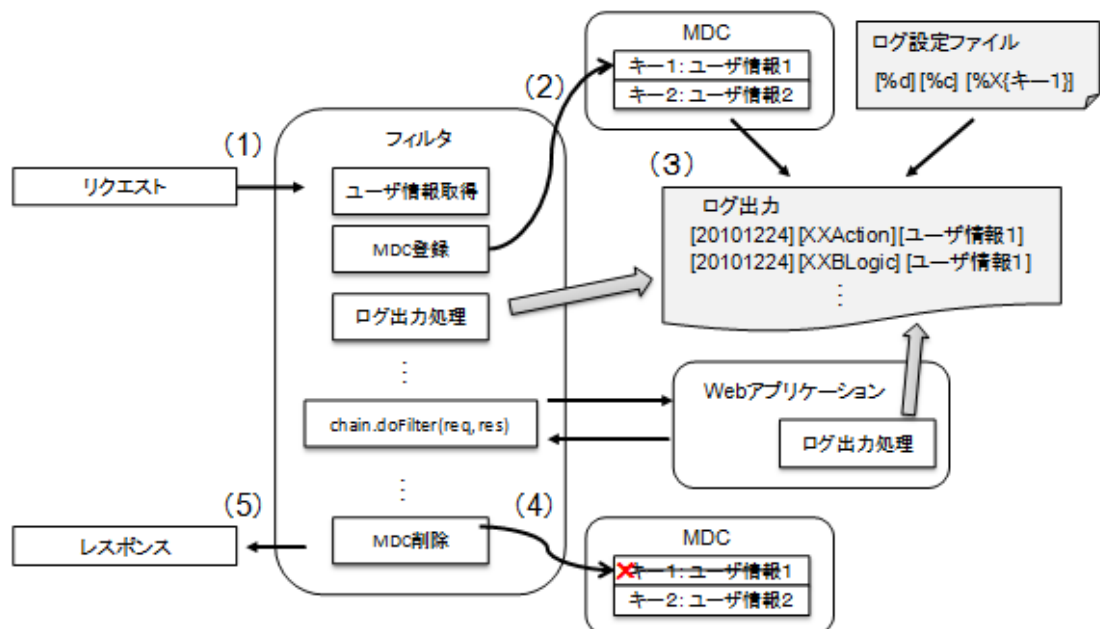
AL009-02 操作ログ出力機能

■ 概要

◆ 機能概要

- 操作するユーザ ID 等をログ出力する方法を解説する
- Terasoluna のユーザ管理機能と Log4J の MDC 機能を利用してログ出力する。

◆ 概念図



◆ 解説

- ユーザからリクエストが発行される
- フィルタ内で、セッション領域内に保持されたユーザ情報を取得し、MDC（マップ化診断コンテキスト）に登録する。
- フィルタもしくはアプリケーション内のログ出力時に、MDC とログ設定ファイルをもとにリクエストを発行したユーザ情報がログ出力される。
- MDC はスレッドローカルスコープのオブジェクトであるため、スレッドプールに返却される前にユーザ情報を削除しておく。
- レスポンスがユーザに返却される。

■ 使用方法

◆ コーディングポイント

- Log4j による MDC（マップ化診断コンテキスト）使用方法

Web アプリケーションにおいて、操作するユーザ ID をログに記録したいことがある。通常は、すべてのログ出力する箇所で、セッションに格納されている UVO からユーザ ID を取得して、置換文字列のパラメータとして渡してログ出力する必要がある。しかし、ログ出力機構として Log4J を採用している場合は、MDC（Mapped Diagnostic Context）を利用することで煩雑な作業を軽減することが可能である。サーブレット Filter にて Log4J の MDC に、セッションのユーザ ID 等を登録しておくことで、後は Log4J 設定ファイルにて自由に値を取得し、出力することができる。

➤ Log4J.MDC 登録の実装クラスサンプル

```
public class LoggerFilter implements Filter {
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
        try {
            // セッションから USER_VALUE_OBJECT を取得
            HttpSession session = ((HttpServletRequest) request).getSession();
            Object UVO = session
                .getAttribute(UserValueObject.USER_VALUE_OBJECT_KEY);

            // UVO が設定されていたら MDC に登録する
            if (UVO != null) {
                MDC.put("USER_ID", ((SampleUVO) UVO).getUserId());
            }
            // 次の Filter 処理を実行する。
            chain.doFilter(request, response);
        } finally {
            // メッセージをマップから削除する。
            // キー：USER_ID
            MDC.remove("USER_ID");
        }
    }
}
```

スレッド単位に保持されるため、スレッドプールに戻る前にログ情報を削除しておくこと

➤ Log4j 設定ファイルサンプル

ログレベル、アペンダ名設定

log4j.rootCategory=DEBUG, consoleLog

コンソールアペンダ設定

log4j.appender.consoleLog=org.apache.log4j.ConsoleAppender

log4j.appender.consoleLog.Target=System.out

log4j.appender.consoleLog.layout=org.apache.log4j.PatternLayout

log4j.appender.consoleLog.layout.ConversionPattern=

[%d{yyyy/MM/dd HH:mm:ss}][%p][%c{1}] [%X{USER_ID}] %m%n

%X{}の{}内に MDC に登録したキーを設定する。

➤ ログファイル出力例

[2010/12/06 18:38:19][DEBUG][ActionEx] [0000001] execute() called.

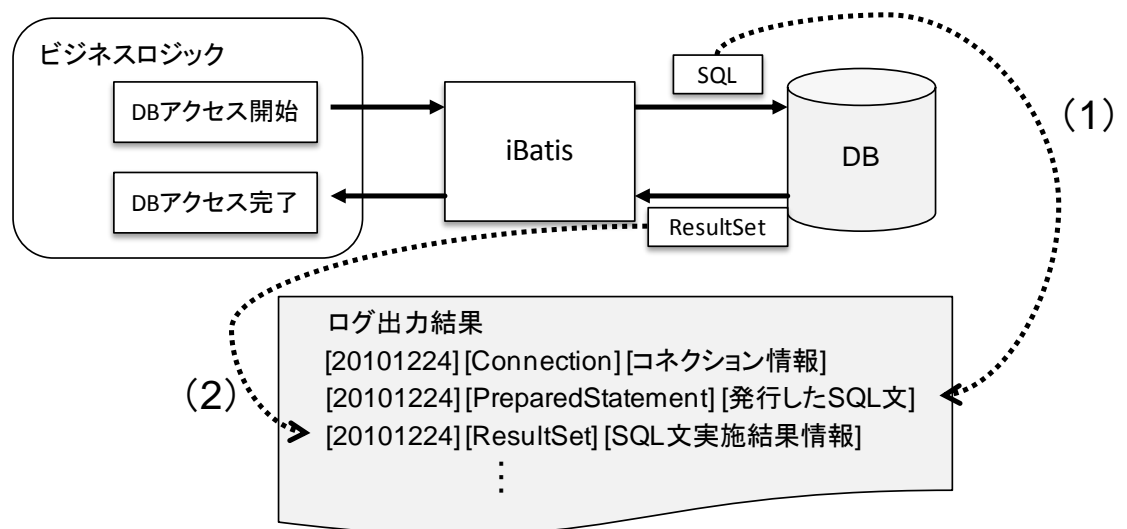
AL009-03 DB アクセスログ出力機能

■ 概要

◆ 機能概要

- 操作するユーザ ID 等をログ出力する方法を解説する
- Terasoluna のユーザ管理機能と Log4J の MDC 機能を利用してログ出力する。

◆ 概念図



◆ 解説

- iBatis から発行された SQL 文がログに出力される
- DB から返却された ResultSet がログに出力される

■ 使用方法

◆ コーディングポイント

- Log4j による iBATIS のデバッグログ出力方法

Web アプリケーションで実行した SQL 文をログで出力したい場合は iBATIS のログを出力するように設定する。iBATIS のログ出力を確認するためには Log4j の設定を以下のように設定する。

➤ Log4j 設定ファイルサンプル

```
# ログレベル、アペンダ名設定
log4j.rootCategory=INFO, consoleLog

# ログレベルの詳細設定
log4j.category.java.sql.Connection= DEBUG
log4j.category.java.sql.Statement= DEBUG
log4j.category.java.sql.PreparedStatement= DEBUG
log4j.category.java.sql.ResultSet= DEBUG
log4j.category.com.ibatis=DEBUG
#log4j.category.com.ibatis.common.jdbc.SimpleDataSource= DEBUG
#log4j.category.com.ibatis.sqlmap.engine.cache.CacheModel= DEBUG
#log4j.category.com.ibatis.sqlmap.engine.impl.SqlMapClientImpl= DEBUG
#log4j.category.com.ibatis.sqlmap.engine.builder.xml.SqlMapParser= DEBUG
#log4j.category.com.ibatis.common.util.StopWatch= DEBUG
```

iBATIS ではなく、java.sql.* のログを出力するように設定する。

(※) iBAITS 内のデバッグログを取得したい場合は、上記設定のコメントアウトを外すこと

対象となるログ出力クラスは以下の通りである。

- java.sql.Connection
- java.sql.PreparedStatement
- java.sql.Resultset
- java.sql.Statement
- com.ibatis

また、iBATIS のログを出力する際は、実行された SQL 文が出力される。どの SQL 文が特定するためには、SQL ID を SQL 文のコメントに含めておく必要がある。

➤ SQLMap 設定例

```
<select id="loginUser" parameterClass="jp.co.nttdata.project.dao.LoginUserInput"
      resultClass="jp.co.nttdata.project.dao.LoginUserOutput">
  /* loginUser */
  SELECT  user_id AS "userId" ,user_name AS "userName"
  FROM user_table WHERE user_id = #userId# AND user_pw = #password#
</select>
```

SQL 文が全てログに出力されるため、SQL 文を特定するために、SQL ID をコメントとして SQL 文に埋め込んでおく。

➤ ログファイル出力例

```
[2010/12/06 18:38:16][DEBUG][Connection] {conn-100006} Connection
[2010/12/06 18:38:16][DEBUG][Connection] {conn-100006}
Preparing Statement: /* loginUser */ SELECT user_id AS "userId",user_name AS
"userName" FROM user_table WHERE user_id = ? AND user_pw = ?
[2010/12/06 18:38:16][DEBUG][PreparedStatement] {pstm-100007}
Executing Statement: /* loginUser */ SELECT user_id AS "userId",user_name AS
"userName" FROM user_table WHERE user_id = ? AND user_pw = ?
[2010/12/06 18:38:16][DEBUG][PreparedStatement] {pstm-100007}
Parameters: [0000001, password1]
[2010/12/06 18:38:16][DEBUG][PreparedStatement] {pstm-100007}
Types: [java.lang.String, java.lang.String]
[2010/12/06 18:38:16][DEBUG][ResultSet] {rset-100008} ResultSet
[2010/12/06 18:38:16][DEBUG][ResultSet] {rset-100008}
Header: [userId, userName]
[2010/12/06 18:38:16][DEBUG][ResultSet] {rset-100008} Result: [0000001, name1]
```

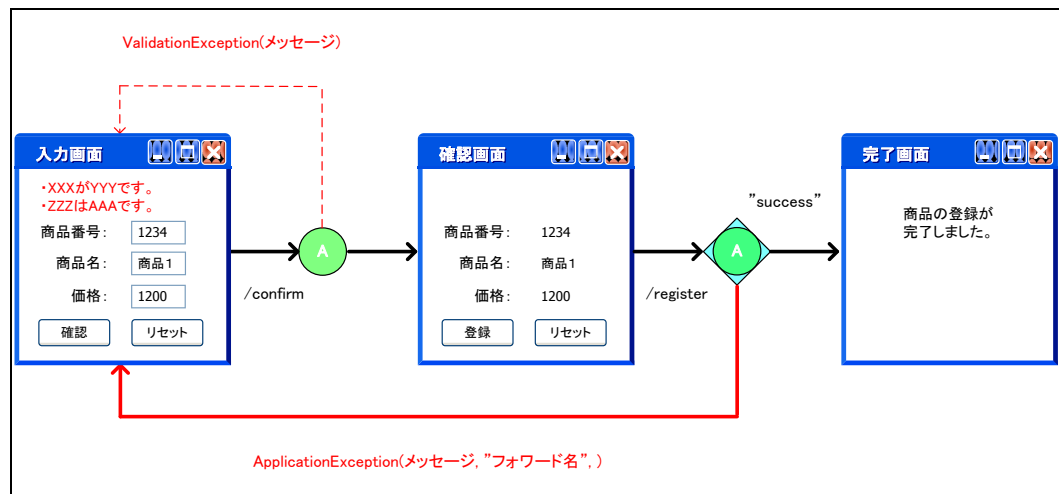
AL010 共通例外機能

■ 概要

◆ 機能概要

- Struts の宣言的例外処理機構を拡張し、頻繁に利用されることが想定される共通的な例外クラス、および例外ハンドラクラスを提供する。
- 本機能は、以下のコンポーネントから構成される。
 - ValidationException
 - ◇ 検証に失敗したことを通知するための例外クラス。
 - ◇ 入力値に不備があり、後続の業務処理が実行できない場合に送出
 - ◇ 画面表示用のエラーメッセージ文字列（BLogicMessages）を指定して throw
 - ValidationExceptionHandler
 - ◇ ValidationException 専用の例外ハンドラクラス
 - ◇ 入力元画面（action の input 属性）に遷移
 - ◇ ValidationException が保持するエラーメッセージ文字列を、Globals.ERROR_KEY で指定されたスコープに登録
 - ApplicationException
 - ◇ 業務処理中に何らかの不都合が発生し、処理が続行できないことを通知するための例外クラス
 - ◇ 画面表示用のエラーメッセージ文字列（BLogicMessages）とフォワード先（省略可）を指定して throw
 - ApplicationExceptionHandler
 - ◇ ApplicationException 専用の例外ハンドラクラス
 - ◇ ApplicationException が保持するフォワード先に遷移
 - ◇ ApplicationException が保持するエラーメッセージ文字列を、Globals.ERROR_KEY で指定されたスコープに登録

◆ 概念図



◆ 解説

- Struts 設定ファイル（struts-config.xml）の global-exceptions 要素下に ValidationExceptionHandler、および ApplicationExceptionHandler を正しく設定することで、各開発者が独自の例外クラスを定義したり、煩雑な例外処理を記述することなく FW が提供する例外処理機能を利用できる。設定方法詳細は、コーディングポイントを参照すること。

■ 使用方法

◆ コーディングポイント

- Struts 設定ファイル（struts-config.xml）に exception 要素を記述
フレームワークが提供する例外処理機能は、Struts 設定ファイル（struts-config.xml）の global-exceptions 要素下に、ValidationExceptionHandler および ApplicationExceptionHandler に関する exception 要素を記述することで利用できる。

➤ Struts 設定ファイル（struts-config.xml） exception 要素

```

<global-exceptions>
  <exception
    key=""
    path="/error.jsp"
    className="jp.terasoluna.fw.web.struts.action.ExceptionConfigEx"
    type="jp.terasoluna.fw.exception.ValidationException"
    handler="jp.terasoluna.fw.ex.web.struts.action.ValidationExceptionHandler">

```

```
</exception>

<exception
  key=""
  path="/error.jsp "
  className="jp.terasoluna.fw.web.struts.action.ExceptionConfigEx"
  type="jp.terasoluna.fw.exception.ApplicationException"
  handler="jp.terasoluna.fw.ex.web.struts.action.ApplicationExceptionHandler">
</exception>
</global-exceptions>
```

上記の設定を行うことで、`ValidationExceptionHandler` は `Action`（および `BLogic`）実行中に送出された全ての `ValidationException` を捕捉し、`ValidationException` が保持するエラーメッセージ文字列を `Globals.ERROR_KEY` で指定されたスコープに登録した後、入力元画面（実行中の `action` 要素の `input` 属性）に遷移する。

なお、実行中の `action` 要素に `input` 属性が定義されていない場合、`exception` 要素の `path` 属性で定義された遷移先に遷移する。

また、`ApplicationExceptionHandler` は `Action`（および `BLogic`）実行中に送出された全ての `ApplicationException` を捕捉し、`ApplicationException` が保持するエラーメッセージ文字列を `Globals.ERROR_KEY` で指定されたスコープに登録した後、以下の優先順位に従って遷移先を解決する。

- ✧ `ApplicationException` にフォワード名が指定されている場合、実行中の `action` 要素下の `forward` 要素一覧（ローカルフォワード）から `name` 属性が一致する `forward` 要素を探し出し、見つかった場合には `path` 属性で定義された遷移先に遷移する。
- ✧ ローカルフォワード一覧に見つからなかった場合、`global-forwards`（グローバルフォワード）の一覧から `forward` 要素を探し出す。見つかった場合には同じく `path` 属性で定義された遷移先に遷移する。
- ✧ `ApplicationException` にフォワード名が指定されていない場合、もしくはローカルフォワード一覧にもグローバルフォワード一覧にも `name` 属性の一致する `forward` 要素が見つからなかった場合、`exception` 要素の `path` 属性で定義された遷移先に遷移する。`exception` 要素の `path` 属性が未指定の場合、入力元画面（実行中の `action` 要素の `input` 属性）に遷移する。

- Struts 設定ファイル (struts-config.xml) に action 要素を記述
実行中に `ValidationException` を送出する可能性がある action 要素には、原則的に `input` 属性を指定しておく。また、`ApplicationException` を送出する可能性がある action 要素には、`ApplicationException` 送出時の遷移先となる `forward` 要素を必要に応じて複数定義しておく。
- Struts 設定ファイル (struts-config.xml) action 要素

```
<action path="/confirm"
  name="_Module1Form"
  scope="session"
  validate="true"
  input="/入力画面.jsp">
  <set-property property="clearForm" value="false" />
  <forward name="success" path="/確認画面.jsp" />
</action>
<action path="/register"
  name="_Module1Form"
  scope="session"
  validate="false"
  input="/確認画面.jsp">
  <set-property property="clearForm" value="false" />
  <forward name="success" path="/確認画面.jsp" />
  <forward name="failure" path="/入力画面.jsp" />
</action>
```

上記の action 要素に該当する Action クラス、BLogic クラスを定義した Bean 定義ファイルも併せて提示しておく。

- Bean 定義ファイル

```
<bean name="/confirm "
  class="jp.terasoluna.fw.web.struts.actions.BLogicAction">
  <property name="tokenCheck" value="false" />
  <property name="saveToken" value="true" />
  <property name="businessLogic" ref="/ConfirmBLogic" />
</bean>
<bean name="/register "
  class="jp.terasoluna.fw.web.struts.actions.BLogicAction">
  <property name="tokenCheck" value="false" />
  <property name="saveToken" value="true" />
  <property name="businessLogic" ref="/RegisterBlogic" />
</bean>
```

- BLogic クラスの作成

入力値に不備があり、入力元画面に遷移させてユーザに入力値の修正を促す場合、その旨を伝えるメッセージ文字列（BLogicMessages）を指定して、ValidationException のインスタンスを生成し、BLogic#execute メソッドから送出する。

- BLogic クラス（ValidationException を送出）

```
public class ConfirmBLogic implements BLogic<InputBean> {

    public BLogicResult execute(InputBean param) {
        String itemName = param.getItemName();
        if ("aaa".equals(itemName)) {
            BLogicMessages bLogicMessages = new BLogicMessages();
            BLogicMessage bLogicMessage = new BLogicMessage(
                "errors.invalidItemName", itemName);
            bLogicMessages.add("GROUP_ERROR", bLogicMessage);
            throw new ValidationException(bLogicMessages);
        }
        // 以下略
    }
}
```

また、業務処理中に何らかの不都合が発生したため、任意の画面に遷移させて処理が続行できないことを通知し、ユーザに対処を促す場合、その旨を伝えるメッセージ文字列（BLogicMessages）とフォワード名を指定して ApplicationException のインスタンスを生成し、BLogic#execute メソッドから送出する。

- BLogic クラス（ApplicationException を送出）

```
public class RegisterBLogic implements BLogic<InputBean> {

    public BLogicResult execute(InputBean param) {
        String itemName = param.getItemName();
        if ("aaa".equals(itemName)) {
            BLogicMessages bLogicMessages = new BLogicMessages();
            BLogicMessage bLogicMessage = new BLogicMessage(
                "errors.invalidItemName", itemName);
            bLogicMessages.add("GROUP_ERROR", bLogicMessage);
            throw new ApplicationException(bLogicMessages, "failure");
        }
        // 以下略
    }
}
```

- エラーメッセージ文字列の表示（JSP）

ValidationException や ApplicationException に設定したメッセージ文字列は、

TERASOLUNA や Struts が提供するカスタムタグを使って取得・画面表示する。
タグの利用方法詳細やサポートする属性は、各フレームワークの API を参照すること。

➤ BLogic クラス (ValidationException を送出)

```
<!-- Struts が提供するカスタムタグ -->  
<html:errors />  
  
<!-- TERASOLUNA が提供するカスタムタグ -->  
<ts:errors />
```

AL017 ビジネスロジック入出力定義ファイル削減機能

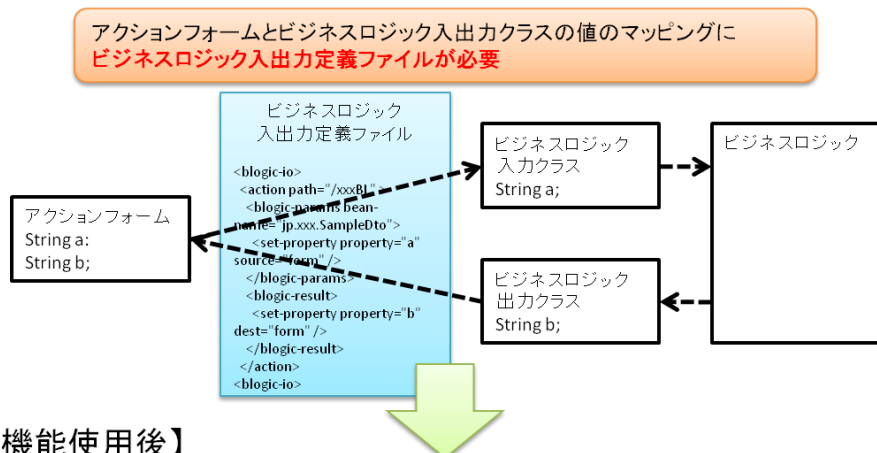
概要

◆ 機能概要

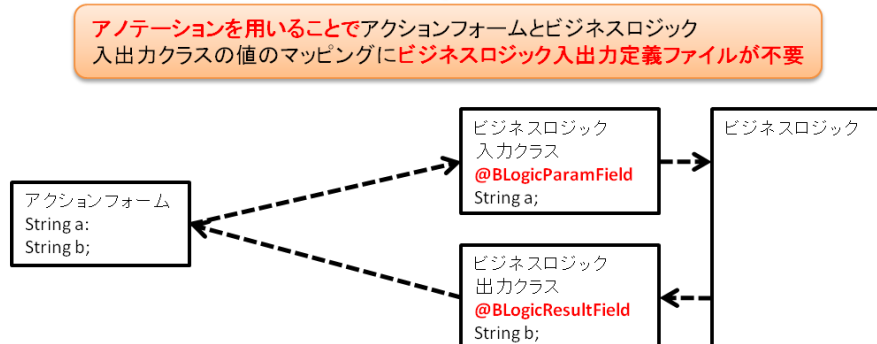
- ビジネスロジック入出力情報定義ファイル(blogic-io.xml)の削減を行う。
- アノテーションに入出力の設定を記述することで、ビジネスロジックが呼ばれた際に、呼ばれたビジネスロジックに対応するビジネスロジック入出力定義を生成する。そして、プレゼンテーション層⇄ビジネスロジック間のデータ交換を行う。
- リクエスト、セッション、アクションフォーム、サーブレットコンテキストからビジネスロジックへの入力処理、ビジネスロジックからリクエスト、セッション、アクションフォームへの出力処理を行える。
- BLogic インタフェースを実装したビジネスロジック、POJO のビジネスロジック双方に対応している。

機能使用イメージ(プレゼンテーション層がアクションフォームである場合)

【機能使用前】



【機能使用后】

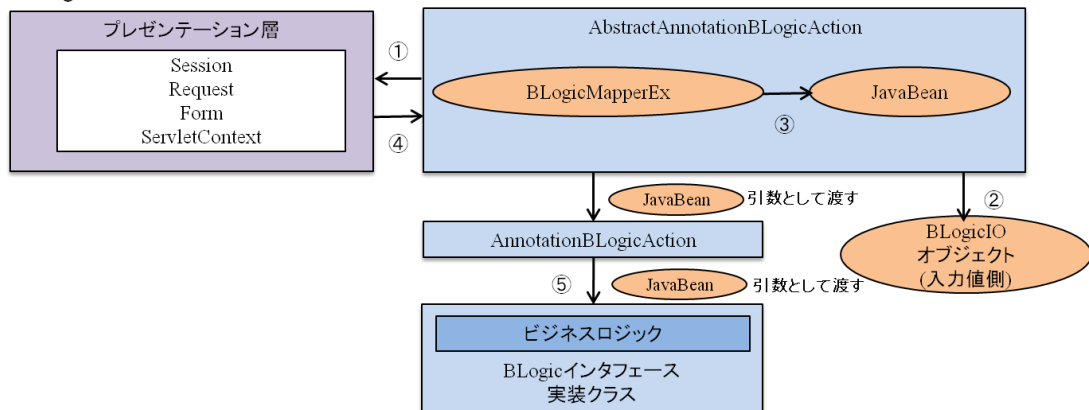


【凡例】 データ -->

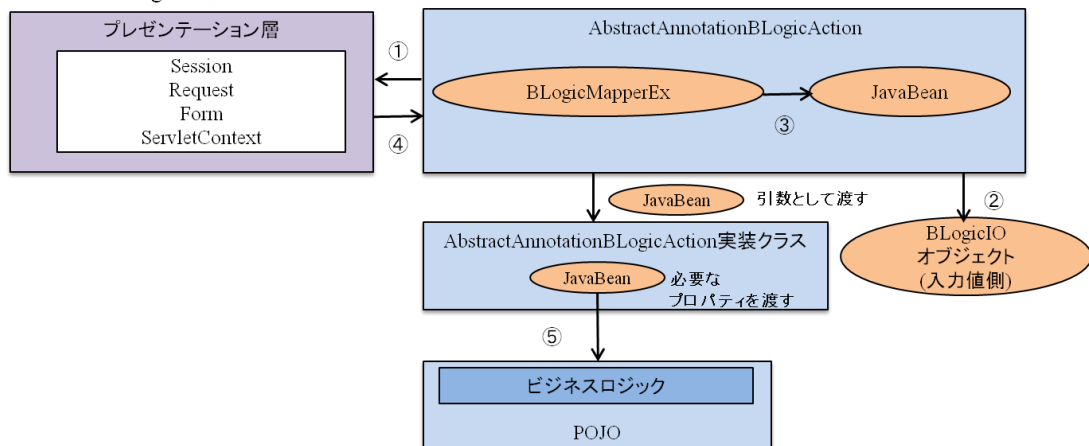
◆ 概念図

➤ ビジネスロジック入力値取得までの流れ

BLogicインタフェースを実装したビジネスロジックの場合



POJOのBLogicの場合



◆ 解説

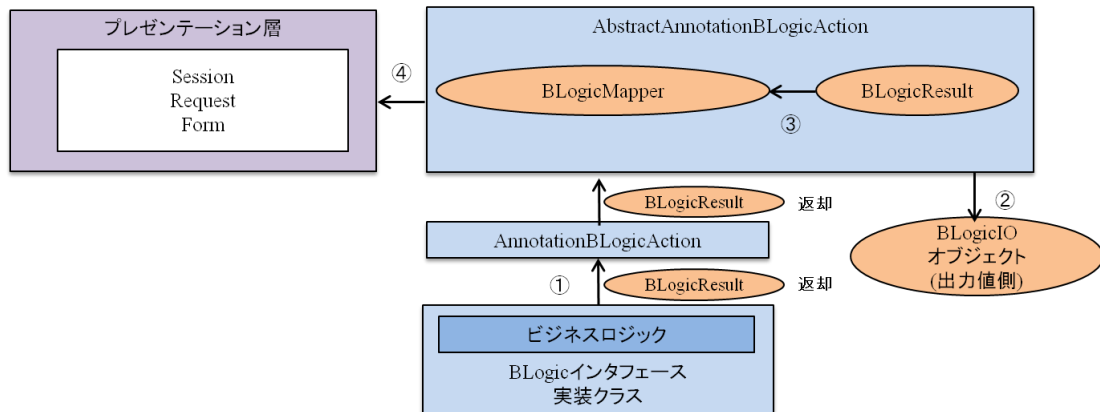
- ① **AbstractAnnotationBLogicAction** はビジネスロジックの入力値オブジェクトを要求する。
- ② **AbstractAnnotationBLogicAction** は呼び出し元のビジネスロジックのビジネスロジック入力クラスのアノテーションから入力値側のビジネスロジック入出力定義 (BLogic-IO オブジェクト) を生成する。
- ③ **BLogicMapperEx** はビジネスロジックの入力値オブジェクトとなる **JavaBean** を生成する。
- ④ **BLogicMapperEx** は②で生成したビジネスロジック入出力定義をもとにプレゼンテーション層の値を入力値オブジェクトに反映させる。
- ⑤ **BLogic** インタフェースを実装したビジネスロジックの場合は **AbstractAnnotationBLogicAction** はビジネスロジックに入力値オブジェクトを渡す。

POJO のビジネスロジックの場合はビジネスロジックに入力値オブジェクトのプロパティを渡す。

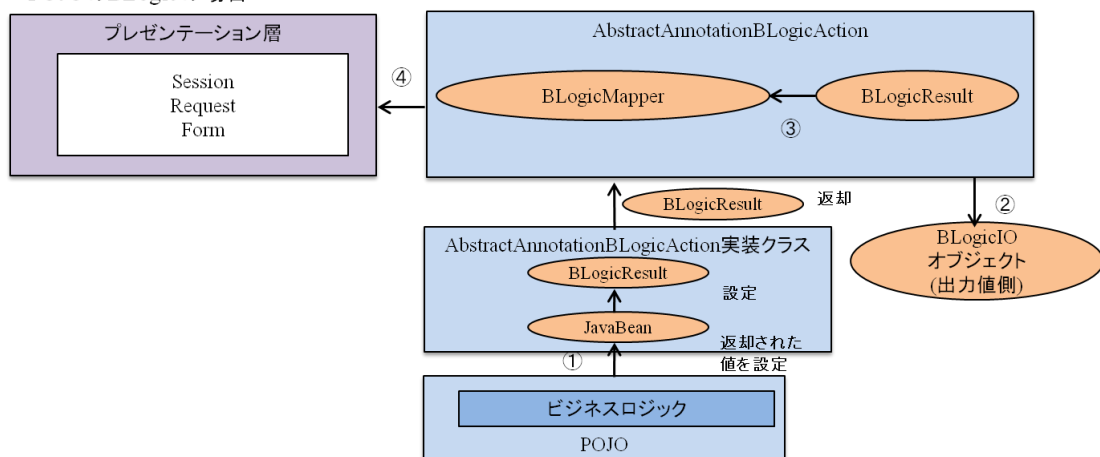
◆ 概念図

➤ ビジネスロジック出力値反映までの流れ

BLogicインタフェースを実装したビジネスロジックの場合



POJOのBLogicの場合



◆ 解説

- ① ビジネスロジックは BLogicResult(POJO の BLogic の場合は JavaBean) を AbstractAnnotationBLogicAction に渡す。
- ② AbstractAnnotationBLogicAction は渡された BLogicResult が持つ ResultObject のアノテーションから出力値側ビジネスロジック入出力定義(BLogic-IO オブジェクト)を生成する。
- ③ AbstractAnnotationBLogicAction は BLogicMapperEx にプレゼンテーション層への反映を依頼する。
- ④ BLogicMapperEx は②で生成したビジネスロジック入出力定義をもとに ResultObject の値をプレゼンテーション層に反映させる。

■ 使用方法

◆ コーディングポイント

- Bean 定義ファイルの設定
本機能を利用するにはビジネスロジックを実行するアクションクラスに、AnnotationBLogicAction を指定する。

➤ Bean 定義ファイルの設定例

```
<bean name="/ABCD_01BL"  
  class="jp.terasoluna.fw.web.struts.actions.AnnotationBLogicAction">  
  <property name="businessLogic" ref="abcd_01BLogic" />  
</bean>
```

- アノテーションについて
 - ビジネスロジック入力値・出力値に関わる定義情報
下記のアノテーションによりビジネスロジック入力値・出力値に関わる定義情報を設定する。
 - ✧ BLogicParamField アノテーション (ビジネスロジック入力値)
 - ✧ BLogicResultField アノテーション (ビジネスロジック出力値)
 - ✧ BLogicIOField アノテーション (ビジネスロジック入力値・出力値)

基本的にはビジネスロジック入力値に関わる定義情報は BLogicParamField アノテーション、ビジネスロジック出力値に関わる定義情報は BLogicResultField アノテーションを用いればよい。

BLogicIOField は下記のような場合に用いることで BLogicParamField や BLogicResultField 用いる場合よりソースコードの記述量を削減可能である。

- ✧ ビジネスロジック入力クラスと出力クラスが同一である
- ✧ プレゼンテーション層からの入力値の取得元とプレゼンテーション層への出力値の反映先が同一である

- アノテーションの優先順位
ビジネスロジック入力クラスの同一の属性に対して BLogicParamField アノテーションと BLogicIOField アノテーションを設定した場合、BLogicParamField アノテーションの設定内容が優先される。

ビジネスロジック出力クラスの同一の属性に対して BLogicResultField アノテーションと BLogicIOField アノテーションを設定した場合、BLogicResultField アノテーションの設定内容が優先される。

- ビジネスロジック入力値に関わる定義情報（BLogicParamField アノテーション）

ビジネスロジック入力値に関わる定義情報は、ビジネスロジック入力クラスの持つ属性に対して BLogicParamField アノテーションにより設定する。

◇ BLogicParamField アノテーションの設定項目

項番	属性名	概要	デフォルト値	省略	
				○:可能 ×:不可能	条件
1	target	プレゼンテーション層からの入力値の取得元を「FORM」「REQUEST」「SESSION」「APPLICATION」の中から指定する。	FORM	○	なし
2	property	プレゼンテーション層での属性名（リクエスト・セッションのキーやアクションフォームのプロパティ名など）を指定する。	なし	○	プレゼンテーション層における属性名が同一の場合

- ビジネスロジック出力値に関わる定義情報（BLogicResultField アノテーション）

ビジネスロジック出力値に関わる定義情報は、ビジネスロジック出力クラスの持つ属性に対して BLogicResultField アノテーションにより設定する。

◇ BLogicResultField アノテーションの設定項目

項番	属性名	概要	デフォルト値	省略	
				○:可能 ×:不可能	条件
1	target	プレゼンテーション層への出力値の反映先を「FORM」「REQUEST」「SESSION」の中から指定する。	FORM	○	なし
2	property	プレゼンテーション層での属性名（リクエスト・セッションのキーやアクションフォームのプロパティ名など）を指定する。	なし	○	プレゼンテーション層における属性名が同一の場合

- ビジネスロジック入力・出力値に関わる定義情報 (BLogicIOField アノテーション)

ビジネスロジック入力・出力値に関わる定義情報は、ビジネスロジック入力クラス・出力クラスの持つ属性に対して BLogicIOField アノテーションにより設定する。

◇ BLogicIOField アノテーションの設定項目

項番	属性名	分類	概要	デフォルト値	省略	
					○:可能 ×:不可能	条件
1	target	入力値	プレゼンテーション層からの入力値の取得元を「FORM」「REQUEST」「SESSION」「APPLICATION」の中から指定する。	FORM	○	なし
		出力値	プレゼンテーション層への出力値の反映先を「FORM」「REQUEST」「SESSION」の中から指定する。	FORM	○	なし
2	property	-	プレゼンテーション層での属性名(リクエスト・セッションのキーやアクションフォームのプロパティ名など)を指定する。	なし	○	プレゼンテーション層における属性名が同一の場合

- ビジネスロジック入力クラス・出力クラスの作成

- BLogicParamField アノテーションを用いた場合のビジネスロジック入力クラスの作成

ビジネスロジック入力クラスは、ビジネスロジッククラスが業務処理を実行するために必要なプレゼンテーション層からの入力値を格納・保持する。プレゼンテーション層からの入力値を保持するプロパティ、および getter・setter を定義し、各属性に対して BLogicParamField アノテーションを付与する。

◇ ビジネスロジック入力クラスの実装例


```
public class ABCD_01Input {  
    // アクションフォームから aaa 属性を取得したい場合  
    @BLogicParamField  
    private String aaa;  
  
    // リクエストから bbb 属性を取得したい場合  
    @BLogicParamField(target = BLogicIOTarget.REQUEST)  
    private String bbb;  
  
    // セッションから ccc 属性を取得したい場合  
    @BLogicParamField(target = BLogicIOTarget.SESSION)  
    private String ccc;  
  
    // アクションフォームから aaa 属性を取得したい場合  
    @BLogicParamField(property = "aaa")  
    private String ddd;  
  
    // アクションフォームから bean プロパティの eee 属性を取得したい場合  
    @BLogicParamField(property = "bean.eee")  
    private String eee;  
  
    // サーブレットコンテキストから ggg 属性を取得したい場合  
    @BLogicParamField(target = BLogicIOTarget.APPLICATION, property="ggg")  
    private String fff;  
  
    // アクションフォームから List<String>型の hhh 属性を取得したい場合  
    @BLogicParamField  
    private List<String> hhh  
  
    // 以下、getter および setter  
}
```

➤ **BLogicResultField** アノテーションを用いた場合のビジネスロジック出力クラスの作成

ビジネスロジック出力値クラスは、ビジネスロジッククラスからの出力値が格納・保持され、プレゼンテーション層に出力値を反映する際に参照される。ビジネスロジッククラスからの出力値を保持する属性および **getter・setter** を定義し、各属性に対して **BLogicResultField** アノテーションを付与する。

◇ ビジネスロジック出力クラスの実装例

```
public class ABCD_01Output {  
    // アクションフォームの aaa 属性に値を設定したい場合  
    @BLogicResultField  
    private String aaa;  
  
    // リクエストの bbb 属性に値を設定したい場合  
    @BLogicResultField(target = BLogicIOTarget.REQUEST)  
    private String bbb;  
  
    // セッションの ccc 属性に値を設定したい場合  
    @BLogicResultField(target = BLogicIOTarget.SESSION)  
    private String ccc;  
  
    // アクションフォームの aaa 属性に値を設定したい場合  
    @BLogicResultField(property = "aaa")  
    private String ddd;  
  
    // アクションフォームの bean プロパティの eee 属性に値を設定したい場合  
    @BLogicResultField(property = "bean.eee")  
    private String eee;  
  
    // アクションフォームの List<String>型に hhh 属性の値を設定したい場合  
    @BLogicResultField  
    private List<String> hhh;  
  
    // 以下、getter および setter  
}
```



target 属性を省略しているためデフォルト値の Form へ値を反映させる。
--

アクションフォームと属性名が一致する場合 property 属性は省略可能

- **BLogicIOField** アノテーションを用いた場合のビジネスロジック入力クラス・出力クラスの作成
アノテーションに**@BLogicIOField** 以外は **BLogicParamField** アノテーション、**BLogicResultField** アノテーションを用いる場合と使用方法は同様である。
- ビジネスロジック入力クラスと出力クラスが同一の場合について
プレゼンテーション層からの入力値の取得元とプレゼンテーション層への出力値の反映先を個別に設定したい場合は、**BLogicParamField** アノテーションと **BLogicResultField** アノテーションを用いる。
◇ ビジネスロジック入力・出力クラスの実装例

```
public class ABCD_01Dto{  
    // リクエストから aaa 属性の値を取得し  
    // セッションへ aaa 属性の値を反映させたい場合  
    @BLogicParamField(target = BLogicIOTarget.REQUEST)  
    @BLogicResultField(target = BLogicIOTarget.SESSION)  
    private String aaa;  
}
```

プレゼンテーション層からの入力値の取得元とプレゼンテーション層への出力値の反映先を個別に設定できる

プレゼンテーション層からの入力値の取得元とプレゼンテーション層への出力値の反映先が同一である場合、**BLogicIOField** アノテーションを用いることで **BLogicParamField** アノテーションと **BLogicResultField** アノテーションを用いる場合より記述量を削減できる。ただし、**BLogicIOField** アノテーションの属性「target」の値には「APPLICATION」を用いることができない点に注意が必要である。

◇ ビジネスロジック入力・出力クラスの実装例

```
public class ABCD_01Dto{  
    // セッションから aaa 属性の値を取得し  
    // セッションへ aaa 属性の値を反映させたい場合  
    @BLogicIOField(target = BLogicIOTarget.SESSION)  
    private String aaa;  
}
```

プレゼンテーション層からの入力値の取得元とプレゼンテーション層への出力値の反映先は同一になる

- ビジネスロジッククラスの作成

- BLogic インタフェースを実装したビジネスロジックの場合

下記は BLogic インタフェースを実装した場合のビジネスロジッククラスの実装例である。『WH-01 ビジネスロジック実行機能』におけるビジネスロジックの作成と異なる箇所はない。

◇ ビジネスロジッククラスの実装例

```
public class Transition01BLogic implements BLogic<ABCD_01Input> {  
    * ビジネスロジックを実行します。  
    * @param params  
    *     ビジネスロジック入力値  
    * @return ビジネスロジック処理結果  
    * @see jp.terasoluna.fw.service.thin.BLogic#execute(java.lang.Object)  
    */  
    public BLogicResult execute(ABCD_01Input params) {  
        --省略--  
        BLogicResult result = new BLogicResult();  
        // ビジネスロジックの処理結果をビジネスロジック出力値オブジェクトに設定  
        ABCD_01Output abcd_01Output = new ABCD_01Output ();  
        --省略--  
        result.setResultObject(abcd_01Output);  
        result.setResultString("success");  
        return result;  
    }  
}
```

ビジネスロジック入力クラスが引数となっている。

ビジネスロジック出力クラスをビジネスロジック結果オブジェクトに設定する。

- POJO のビジネスロジックの場合

『WH-01 ビジネスロジック実行機能』を参考にして作成すること。注意点として『WH-01 ビジネスロジック実行機能』では、AbstractBLogicAction を拡張した Action クラスを作成しているが、本機能では AbstractAnnotationBLogicAction を拡張した Action クラスを作成する必要がある。

- BLogicMapperEx について

本機能を用いる場合、TERASOLUNA が提供する BLogicMapper を BLogicMapperEx に差し替えることが望ましい。

BLogicMapperEx は、ビジネスロジック入力値のプレゼンテーション層からの取得、ビジネスロジック出力値のプレゼンテーション層への反映を行うが、下記の点が BLogicMapper と異なる。

- 入力値をリクエストパラメータから取得可能
ビジネスロジック入力値を `HttpServletRequest#getAttribute(String)` で取得できない場合、`HttpServletRequest#getParameterValues(String)` を実行し、リクエストパラメータから値を取得している。
実行結果が `null` の場合は `null` を、単一パラメータの場合はその値を表す文字列を、複数パラメータの場合はパラメータ値を保持する文字列配列を返す。
- Form・Request への出力値反映の際に、`null` を許すかどうかを選択可能
システムのプロパティに「`blogicMapperEx.nullIsNotSet=true`」と設定する事によりビジネスロジック出力値を Form および Request に設定する際の挙動を、出力値が `null` の場合にはプレゼンテーション層に値を設定しないように変更できる。
デフォルトは `false` になっており、ビジネスロジック出力値が `null` の場合、Form、Request の属性にそのまま `null` を設定する（TERASOLUNA 標準の BLogicMapper クラスと同じ挙動を取る）。
- resources プロパティへの値の設定が不要
BLogicIOPlugIn の resources プロパティへの値の設定が不要である。

BLogicMapperEx は Struts 設定ファイルにて BLogicIOPlugIn の `mapperClass` プロパティにクラス名を完全修飾で指定することにより利用できる。

- Struts 設定ファイルの設定例 (struts-config.xml)

```
<plug-in className="jp.terasoluna.fw.web.struts.plugins.BLogicIOPlugIn">  
  <set-property property="mapperClass"  
    value="jp.terasoluna.fw.service.thin.BLogicMapperEx " />  
</plug-in>
```

●
<set-property
property="resources" value="～">が不要

BLogicMapperEx の挙動が要件に合致しない場合、`mapperClass` プロパティに TERASOLUNA 標準提供の BLogicMapper クラスを指定し、利用することも可能である。その場合、`resources` プロパティの `value` 属性には何らかの値を設定する必要がある。※2（例えば、空の業務入出力定義ファイル（`blogic-io.xml`）など）。以下は、TERASOLUNA 標準提供の BLogicMapper クラスを利用する場合の Struts 設定ファイル、および空の入出力設定ファイル（`blogic-io.xml`）の例である。

➤ Struts 設定ファイルの設定例（`struts-config.xml`）

```
<plug-in className="jp.terasoluna.fw.web.struts.plugins.BLogicIOPlugIn">
  <set-property property="resources" value="/WEB-INF/blogic-io.xml" />
  <set-property property="mapperClass"
    value="jp.terasoluna.fw.service.thin.BLogicMapper" />
</plug-in>
```

➤ 空の業務入出力定義ファイルの設定例（`blogic-io.xml`）

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE logic-io PUBLIC "-//NTTDATA/DTD TERASOLUNA for Spring
logic-io 1.0/JA" "../dtd/blogic-io.dtd">

<blogic-io/>
```

※2 TERASOLUNA 標準の BLogicMapper クラスは、`resources` に値が設定されていない場合 `IllegalArgumentException` を送出する。

● 本機能を用いる場合の制限事項

制限事項は下記のとおりである。

- ビジネスロジック入出力情報定義ファイルを一部に適用してプレゼンテーション層⇔ビジネスロジック間の入出力の設定をすることはできなくなる。
- 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において Map を用いることはできない。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.service.thin.BLogicMapperEx	デフォルトの BLogicMapper クラスを拡張し、入力値の取得元や出力値が null の場合の挙動などを変更できるようにしたクラス。
2	jp.terasoluna.fw.service.thin.BLogicParamField	プレゼンテーション層からの入力値を保持するフィールドであることを示すアノテーション。ビジネスロジック入出力定義情報を作成する際は BLogicIOField よりも優先される。
3	jp.terasoluna.fw.service.thin.BLogicResultField	プレゼンテーション層への出力値を保持するフィールドであることを示すアノテーション。ビジネスロジック入出力定義情報を作成する際は BLogicIOField よりも優先される。
4	jp.terasoluna.fw.service.thin.BLogicIOField	プレゼンテーション層からの入力値、およびプレゼンテーション層への出力値を保持するフィールドであることを示すアノテーション。
5	jp.terasoluna.fw.service.thin.BLogicIOTarget	プレゼンテーション層からの入力値の取得元、およびプレゼンテーション層への出力値の反映先を定義した列挙型。
6	jp.terasoluna.fw.service.thin.BLogicIOFieldBean	ビジネスロジックのフィールドにつけられたアノテーションの情報を保持するクラス。
7	jp.terasoluna.fw.service.thin.BLogicIOUtil	BLogicIO オブジェクトを構築するためのユーティリティクラス。
8	jp.terasoluna.fw.web.struts.actionsAbstractAnnotationBLogicAction	TERASOLUNA 標準の AbstractBLogicAction クラスを拡張した、ビジネスロジック入出力定義のアノテーションでの設定に対応したビジネスロジック起動抽象アクションクラス。
9	jp.terasoluna.fw.web.struts.actionsAnnotationBLogicAction	AbstractAnnotationBLogicAction を拡張したビジネスロジック起動アクションクラス。

■ 関連機能

- 『WH-02 ビジネスロジック入出力機能』

AL018 設定ファイルワイルドカード指定読み込み機能

■ 概要

◆ 機能概要

- ワイルドカードで指定したパスにマッチする Terasoluna の各種設定ファイルを読み込む機能を提供する。
- 対象の設定ファイルは以下の 5 種である。
 - Struts 設定ファイル (struts-config.xml)
 - Bean 定義ファイル(moduleContext.xml)
 - 入力チェック設定ファイル (validation.xml)
 - 入力チェックルール設定ファイル (validation-rules.xml)
 - リセット定義ファイル (reset.xml)

■ 使用方法

◆ コーディングポイント

- ワイルドカードでのパスの記述方法
ワイルドカードの指定方法は「?」「*」「**」の 3 種を状況により使い分ける必要がある、以下に各文字の意味を図示する。

ワイルドカード	説明
「?」	任意の 1 文字にマッチ。
「*」	0 文字以上の任意の文字列にマッチ。
「**」	任意の深さのあらゆるディレクトリやファイル名にマッチ。

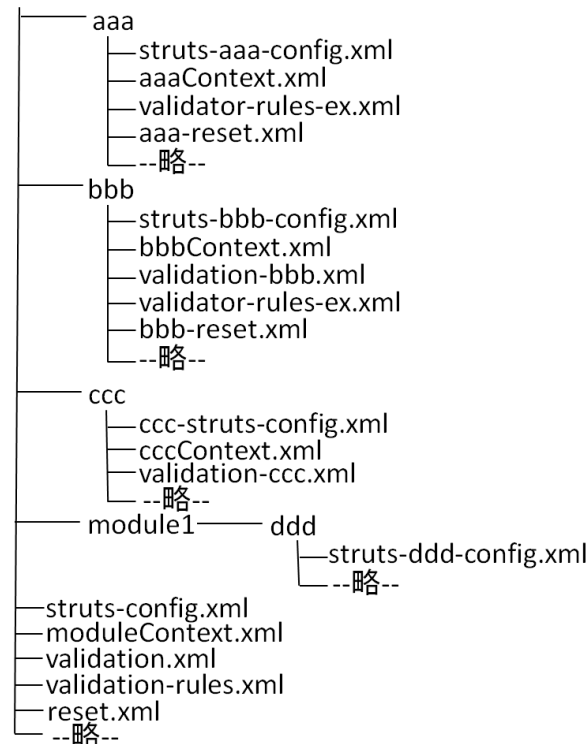
「*」「**」の使い分けには注意が必要である。

例えば「/WEB-INF/*/Sample.xml」と記述すると、WEB-INF から階層が 1 階層下の Sample.xml を指定することを意味する。

しかし「/WEB-INF/**/Sample.xml」と記述すると WEB-INF 直下の Sample.xml および、WEB-INF より 1 階層下以降の Sample.xml を指定することを意味している。

- ワイルドカードを利用した Terasoluna の各種設定ファイルの読み込み
 - 前提となるファイル・フォルダ構成
ファイル・フォルダ構成は以下のようにになっているものとする。

WEB-INF



➤ Struts 設定ファイル (struts-config.xml) の読み込み

Struts 設定ファイルをワイルドカード指定で読み込む場合、Struts が標準で提供する ActionServlet を拡張した ActionServletEx を利用するように Web アプリケーション設定ファイル(web.xml)を記述する。

✧ Web アプリケーション設定ファイルの設定例

```

<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>jp.terasoluna.fw.web.struts.action.ActionServletEx</servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/**/struts*config.xml</param-value>
  </init-param>
  <!-- 以下略 -->
</servlet>
  
```

デフォルトモジュール

本設定により、ActionServletEx は Web アプリケーション起動時に以下の Struts 設定ファイルを読み込み ServletContext に登録する。

/WEB-INF/struts-config.xml

/WEB-INF/aaa/struts-aaa-config.xml

/WEB-INF/bbb/struts-bbb-config.xml

/WEB-INF/module1/ddd/struts-ddd-config.xml

以下の点に注意すること。

- ✧ ワイルドカードの指定に一致しないため `ccc-struts-config.xml` は読み込まれていない
- ✧ `WEB-INF` 直下の `struts-config.xml` が読み込まれている
- ✧ `WEB-INF` より 2 階層下の `struts-ddd-config.xml` が読み込まれている

➤ モジュール分割への対応

本機能は **Struts** のモジュール分割に対応している。以下のようにモジュール分割を設定した場合、デフォルトモジュールとして以下のファイルが読み込まれる。

`/WEB-INF/struts-config.xml`

また、`module1` として以下のファイルが読み込まれる。

`/WEB-INF/module1/ddd/struts-ddd-config.xml`

- ✧ Web アプリケーション設定ファイルの設定例

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>jp.terasoluna.fw.web.struts.action.ActionServletEx</servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts*config.xml</param-value>
  </init-param>
  <init-param>
    <param-name>config/module1</param-name>
    <param-value>/WEB-INF/module1/**/struts*config.xml</param-value>
  </init-param>
</servlet>
```

The diagram illustrates the mapping of Struts configuration files to servlet init-parameters. A box labeled "デフォルトモジュール" (Default Module) points to the first `<param-value>` (`/WEB-INF/struts*config.xml`). A box labeled "module1" points to the second `<param-value>` (`/WEB-INF/module1/**/struts*config.xml`).

➤ Bean 定義ファイル (`moduleContext.xml`) の読み込み。

Bean 定義ファイルの読み込みもワイルドカードで指定することが可能である。

- ✧ Struts 設定ファイルの設定例

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
  <set-property property="contextConfigLocation"
    value="/WEB-INF/moduleContext.xml,/WEB-INF/**/*Context.xml"/>
</plug-in>
```

本設定により、`ContextLoaderPlugIn` は Web アプリケーション起動時に以下の Bean 定義ファイルを読み込み `ServletContext` に登録する。

`/WEB-INF/moduleContext.xml`
`/WEB-INF/aaa/aaaContext.xml`
`/WEB-INF/bbb/bbbContext.xml`
`/WEB-INF/ccc/cccContext.xml`

➤ 入力チェック設定ファイル (`validation.xml`) の読み込み

入力チェック設定ファイル（validation.xml）をワイルドカード指定で読み込む場合、Struts が標準で提供する ValidatorPlugIn を拡張した ValidatorPlugInEx を利用するように Struts 設定ファイルを記述する。

◇ Struts 設定ファイルの設定例

```
<plug-in className="jp.terasoluna.fw.web.struts.plugins.ValidatorPlugInEx">
  <set-property property="pathnames"
    value="/WEB-INF/**/validator-rules*.xml,/WEB-INF/**/validation*.xml"/>
</plug-in>
```

本設定により、ValidatorPlugInEx は Web アプリケーション起動時に以下の入力チェック設定ファイルを読み込み ServletContext に登録する。

/WEB-INF/validator-rules.xml

/WEB-INF/validation.xml

/WEB-INF/aaa/validator-rules-ex.xml

/WEB-INF/bbb/validation-bbb.xml

/WEB-INF/ccc/validation-ccc.xml

➤ リセット定義ファイル（reset.xml）の読み込み

リセット定義ファイル（reset.xml）をワイルドカード指定で読み込む場合、TERASOLUNA が標準で提供する ResetterPlugIn を拡張した ResetterPlugInEx を利用するように Struts 設定ファイル（struts-config.xml）を記述する。

◇ Struts 設定ファイルの設定例

```
<plug-in className="jp.terasoluna.fw.web.struts.plugins.ResetterPlugInEx">
  <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>
```

本設定により、ResetterPlugIn は Web アプリケーション起動時に以下のリセット定義ファイルを読み込み ServletContext に登録する。

/WEB-INF/reset.xml

/WEB-INF/aaa/aaa-reset.xml

/WEB-INF/bbb/bbb-reset.xml

※この際にリセットルール定義ファイル(reset-rules.xml)をリセット定義として読み込まないように、ワイルドカードの指定に注意する事。

- 読み込まれた設定ファイル一覧の確認

読み込まれた設定ファイルの一覧は、アプリケーションサーバの起動時にログ出力される。このログを参照することで、実際に読み込まれた設定ファイルを確認できる。

➤ Web アプリケーション起動ログの例

```
----- 略 -----  
[2011/03/30 14:59:41][INFO][ActionServletEx] Loading Struts module configurations  
from /WEB-INF/aaa/struts-aaa-config.xml  
[2011/03/30 14:59:41][INFO][ActionServletEx] Loading Struts module configurations  
from /WEB-INF/bbb/struts-bbb-config.xml  
[2011/03/30 14:59:41][INFO][ActionServletEx] Loading Struts module configurations  
from /WEB-INF/struts-config.xml  
  
----- 略 -----  
  
[2011/03/30 14:59:41][INFO][XmlBeanDefinitionReader] Loading XML bean  
definitions from ServletContext resource [/WEB-INF/moduleContext.xml]  
[2011/03/30 14:59:41][INFO][XmlBeanDefinitionReader] Loading XML bean  
definitions from ServletContext resource [/WEB-INF/aaa/aaaContext.xml]  
[2011/03/30 14:59:41][INFO][XmlBeanDefinitionReader] Loading XML bean  
definitions from ServletContext resource [/WEB-INF/bbb/bbbContext.xml]  
  
----- 略 -----  
  
[2011/03/30 14:59:41][INFO][ResetterPlugInEx] Loading reset definition file from  
'ServletContext resource [/WEB-INF/reset.xml]'  
[2011/03/30 14:59:41][INFO][ResetterPlugInEx] Loading reset definition file from  
'ServletContext resource [/WEB-INF/aaa/aaa-reset.xml]'  
[2011/03/30 14:59:41][INFO][ResetterPlugInEx] Loading reset definition file from  
'ServletContext resource [/WEB-INF/bbb/bbb-reset.xml]'  
[2011/03/30 14:59:41][INFO][ValidatorPlugInEx] Loading validation rules file from  
'ServletContext resource [/WEB-INF/validator-rules.xml]'  
[2011/03/30 14:59:41][INFO][ValidatorPlugInEx] Loading validation rules file from  
'ServletContext resource [/WEB-INF/validator-rules-ex.xml]'  
[2011/03/30 14:59:41][INFO][ValidatorPlugInEx] Loading validation rules file from  
'ServletContext resource [/WEB-INF/aaa/validation-aaa.xml]'  
[2011/03/30 14:59:41][INFO][ValidatorPlugInEx] Loading validation rules file from  
'ServletContext resource [/WEB-INF/bbb/validation-bbb.xml]'  
  
----- 略 -----
```

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.struts.action.ActionServletEx	デフォルトの <code>ActionServlet</code> クラスを拡張し、ワイルドカードで指定された <code>Struts-config</code> ファイルを読み込む機能を追加したクラス。
2	jp.terasoluna.fw.web.struts.plugins.ValidatorPlugInEx	<code>Struts</code> が提供する <code>ValidatorPlugIn</code> クラスを拡張したプラグイン。 従来のものと比べ、入力値検証定義をワイルドカードで読み込む事が可能。
3	jp.terasoluna.fw.web.struts.plugins.ResetterPlugInEx	<code>PlugIn</code> インターフェイスを実装したアクションフォームプロパティリセット機能に関わるプラグイン、 <code>TERASOLUNA</code> 標準のものと比べて、リセット設定ファイルをワイルドカードで読み込む事が可能。

■ 関連機能

- 『WB-04 フォームプロパティリセット機能』
- 『WF-01 拡張入力チェック機能』

AL020 RequestProcessor 拡張性向上機能

■ 概要

◆ 機能概要

- TERASOLUNA が提供する RequestProcessorEx を利用した場合、入力値の検証処理や、アクションの解決処理の一部分だけを拡張したい場合でも、RequestProcessorEx の拡張が必要であった。
- RequestProcessorEx クラスを拡張した DelegatingRequestProcessorEx クラスを使用する事により、拡張したい部分のみハンドラクラスに処理を委譲する事が可能となる。よって RequestProcessorEx の拡張クラスを作成する事無く個別拡張が可能となる。
- 拡張の対象は RequestProcessorEx クラスの以下のメソッドである。

メソッド名	説明
processMultipart	マルチパートリクエストの際の処理
processActionForm	アクションフォーム取得処理
processPopulate	アクションフォームへのリクエストパラメータ反映処理
processValidate	入力値の検証処理
getDelegateAction	アクションの解決処理

◆ コーディングポイント

- struts-config.xml の設定
struts-config.xml では、RequestProcessor を DelegatingRequestProcessorEx に設定する。
➤ struts-config.xml の設定例

```
<!-- ===== コントローラ定義 -->  
<controller processorClass="jp.terasoluna.fw.web.struts.action. DelegatingRequestProcessorEx ">
```

- applicationContext.xml への拡張ハンドラクラスの定義
通常の Bean を定義する際と同様に、拡張 Handler クラスを定義する。
コンテナから各ハンドラを探す際、DelegatingRequestProcessorEx クラスの中で定義されているデフォルトのハンドラの Bean 名で Bean を検索する。
通常、ハンドラを定義する際はデフォルトのハンドラの Bean 名で各ハンドラを定義する。デフォルトのハンドラ名は以下の通りである。

➤ デフォルトのハンドラ名

メソッド名	インタフェース	デフォルト Bean 名
processMultipart	ProcessMultipartHandler	processMultipartHandler
processActionForm	ProcessActionFormHandler	processActionFormHandler
processPopulate	ProcessPopulateHandler	processPopulateHandler
processValidate	ProcessValidateHandler	processValidateHandler
getDelegateAction	DelegateActionHandler	delegateActionHandler

➤ applicationContext.xml の設定例

<!-- ActiveRequestProcessor の拡張モジュール -->

<bean id="processMultipartHandler"
class="ProcessMultipartHandler インタフェースを実装した拡張クラス"/>

完全修飾のクラス名で記述する。

<bean id="processActionFormHandler"
class="ProcessActionFormHandler インタフェースを実装した拡張クラス"/>

<bean id="processPopulateHandler"
class="ProcessPopulateHandler インタフェースを実装した拡張クラス"/>

<bean id="processValidateHandler"
class="ProcessValidateHandler インタフェースを実装した拡張クラス"/>

<bean id="delegateActionHandler"
class="DelegateActionHandler インタフェースを実装した拡張クラス"/>

● デフォルトのハンドラの Bean 名を変更する方法

system.properties ファイル中に各種ハンドラ毎に Bean 名を変更する事が可能である。

➤ system.properties の設定例

```
#-----
# DelegatingRequestProcessorEx クラスが検索するハンドラの Bean 名

delegatingRequestProcessorEx.processMultipartHandler=modifiedProcessMultipartHandler
delegatingRequestProcessorEx.processActionFormHandle=modifiedProcessMultipartHandler
delegatingRequestProcessorEx.processPopulateHandler=modifiedProcessPopulateHandler
delegatingRequestProcessorEx.processValidateHandler=modifiedProcessValidateHandler
delegatingRequestProcessorEx.delegateActionHandler=modifiedDelegateActionHandler
```

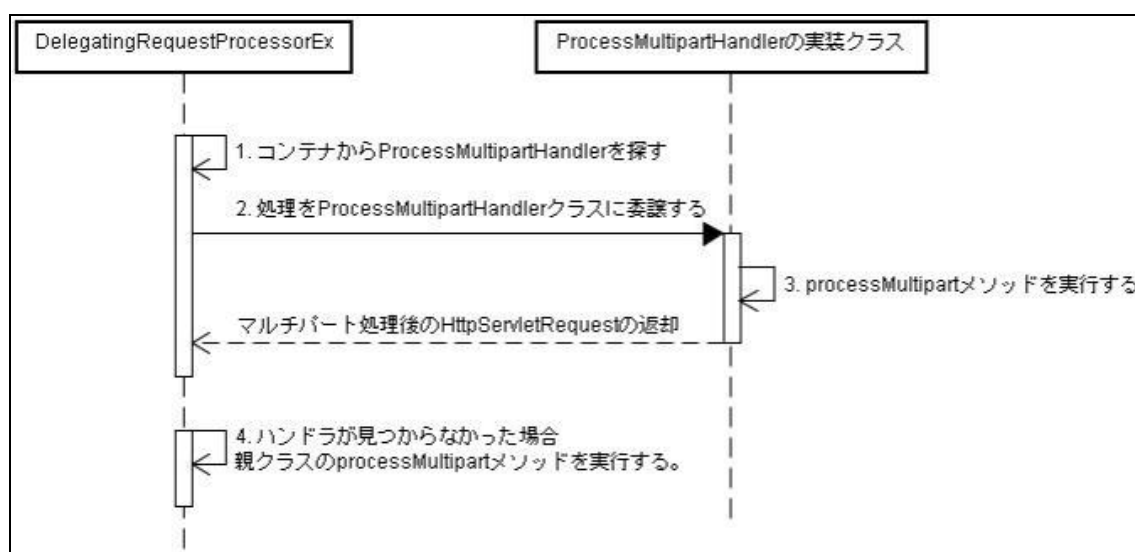
以上のように設定する事で、DelegatingRequestProcessorEx クラスは変更後の Bean 名で各種ハンドラを検索するようになる。

◆ 拡張ポイント

- processMultipart メソッドの拡張を行いたい場合
processMultipart メソッドの拡張処理を定義した、ProcessMultipartHandler インタフェースの実装クラスを作成する。
applicationContext.xml に実装クラスの Bean 定義を記述する。(その際の Bean 名などはコーディングポイントの項目を参照)

拡張処理が行われるタイミングは、下記シーケンス図の 3.の部分となる。

➤ シーケンス図 (processMultipart メソッド)



➤ 解説(processMultipart メソッド)

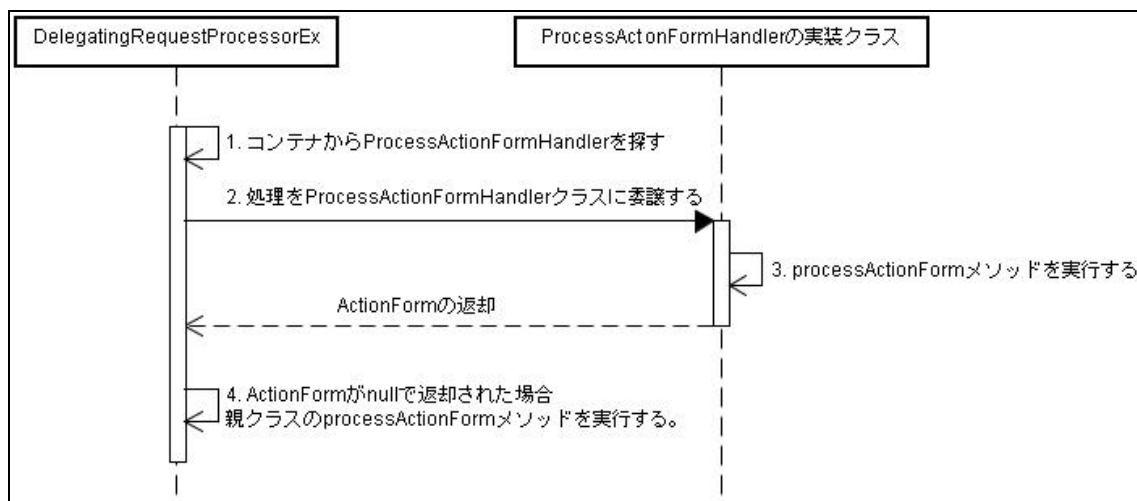
DelegatingRequestProcessorEx#processMultipart メソッドの処理

1. コンテナから ProcessMultipartHandler を探す。
2. ProcessMultipartHandler が発見できた場合は、
DelegatingRequestProcessorEx クラスは ProcessMultipartHandler に処理を委譲する。
3. ProcessMultipartHandler クラスは拡張された processMultipart メソッドを実行し、マルチパート処理後の HttpServletRequest を返却する。
ハンドラが存在した場合は、これ以降の処理を行わず、呼び出し元に **HttpServletRequest** を返却する。
4. ハンドラが存在しなかった場合は親クラスの processMultipart メソッドを実行する。

- processActionForm メソッドの拡張を行いたい場合
processActionForm メソッドの拡張処理を定義した、ProcessActionFormHandler インタフェースの実装クラスを作成する。
applicationContext.xml に実装クラスの Bean 定義を記述する。(その際の Bean 名などはコーディングポイントの項目を参照)

拡張処理が行われるタイミングは、下記シーケンス図の 3.の部分となる。

➤ シーケンス図(processActionForm メソッド)



➤ 解説(processActionForm メソッド)

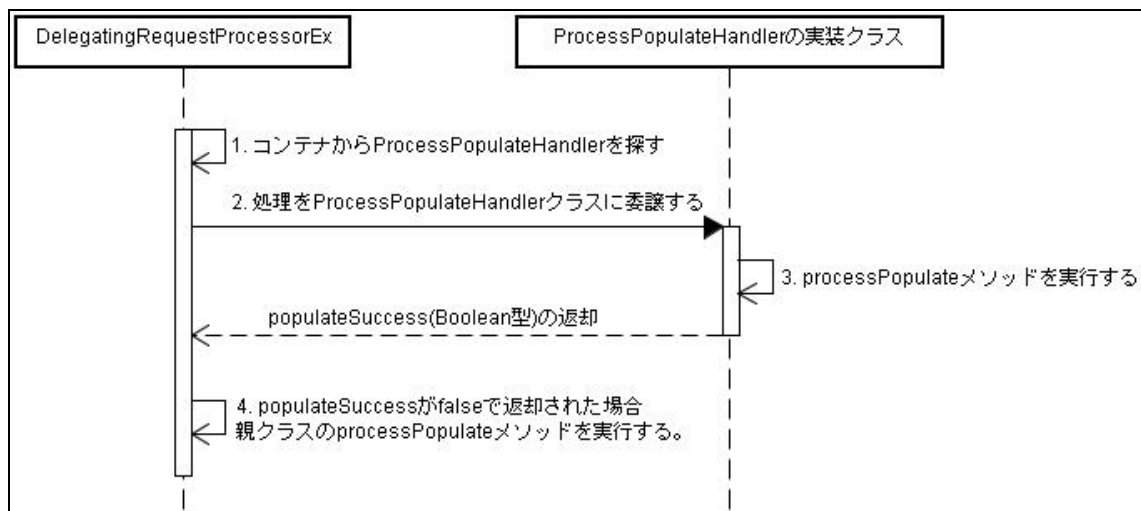
DelegatingRequestProcessorEx# processActionForm メソッドの処理

1. コンテナから ProcessActionFormHandler を探す。
2. ProcessActionFormHandler が発見できた場合は、
DelegatingRequestProcessorEx クラスは ProcessActionFormHandler に処理を委譲する。
3. ProcessActionFormHandler クラスは拡張された processActionForm メソッドを実行し、ActionForm を返却する。
4. 返却された ActionForm が null だった場合は、
DelegatingRequestProcessorEx クラスは親クラスの processActionForm メソッドを実行し、ActionForm を取得する。

- processPopulate メソッドの拡張を行いたい場合
processPopulate メソッドの拡張処理を定義した、ProcessPopulateHandler インタフェースの実装クラスを作成する。
applicationContext.xml に実装クラスの Bean 定義を記述する。(その際の Bean 名などはコーディングポイントの項目を参照)

拡張処理が行われるタイミングは、下記シーケンス図の 3.の部分となる。

➤ シーケンス図(processPopulate メソッド)



➤ 解説(processPopulate メソッド)

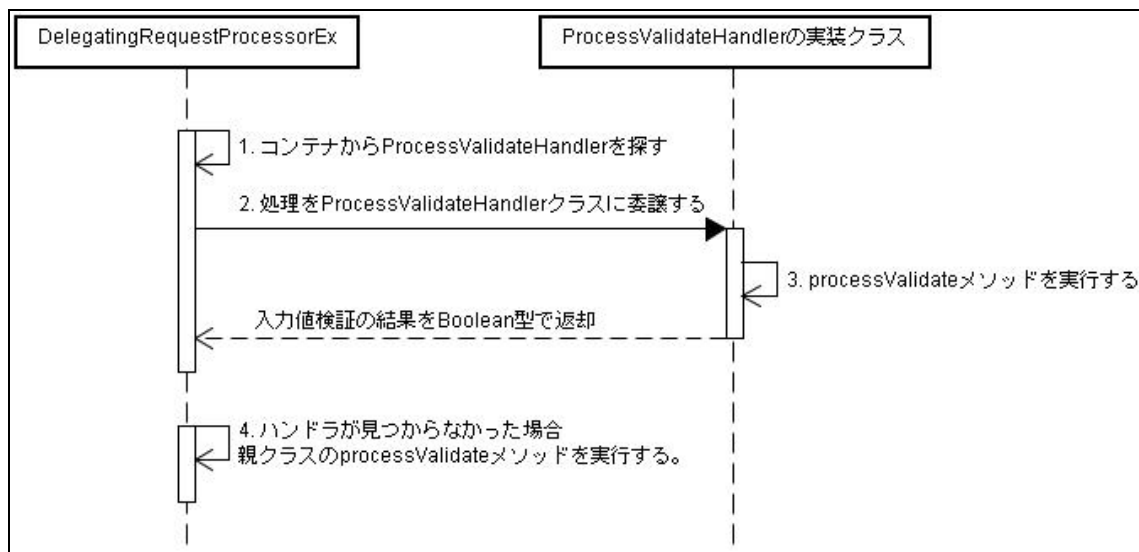
DelegatingRequestProcessorEx# processPopulate メソッドの処理

1. コンテナから ProcessPopulateHandler を探す。
2. ProcessPopulateHandler が発見できた場合は、
DelegatingRequestProcessorEx クラスは ProcessPopulateHandler に処理を委譲する。
3. ProcessPopulateHandler クラスは拡張された processPopulate メソッドを実行し、populateSuccess(Boolean 型)を返却する。
4. 返却された populateSuccess が false だった場合は、
DelegatingRequestProcessorEx クラスは親クラスの processPopulate メソッドを実行し、populateSuccess を取得する。

- processValidate メソッドの拡張を行いたい場合
processValidate メソッドの拡張処理を定義した、ProcessValidateHandler インタフェースの実装クラスを作成する。
applicationContext.xml に実装クラスの Bean 定義を記述する。(その際の Bean 名などはコーディングポイントの項目を参照)

拡張処理が行われるタイミングは、下記シーケンス図の 3.の部分となる。

➤ シーケンス図(processValidate メソッド)



➤ 解説(processValidate メソッド)

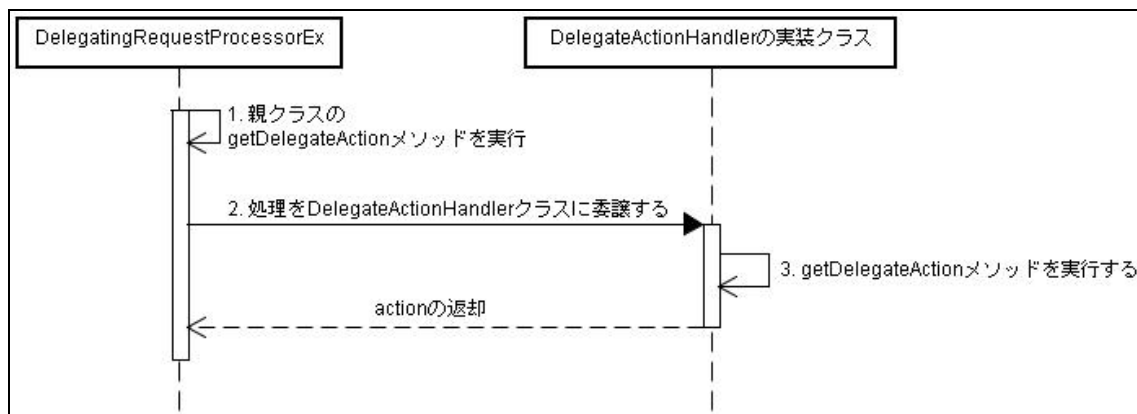
DelegatingRequestProcessorEx# processValidate メソッドの処理

1. コンテナから ProcessValidateHandler を探す。
2. ProcessValidateHandler が発見できた場合は、DelegatingRequestProcessorEx クラスは ProcessValidateHandler に処理を委譲する。
3. ProcessValidateHandler クラスは拡張された processValidate メソッドを実行し、入力値検証結果を Boolean 型で返却する。
ハンドラが存在した場合は、拡張された **processValidate** メソッドの結果に関わらず、これ以降の処理を行わず、呼び出し元に拡張処理の結果を返却する。
4. ハンドラが存在しなかった場合は親クラスの processValidate メソッドを実行する。

- `getDelegateActionHandler` メソッドの拡張を行いたい場合
`getDelegateActionHandler` メソッドの拡張処理を定義した、`DelegateActionHandler` インタフェースの実装クラスを作成する。
`applicationContext.xml` に実装クラスの Bean 定義を記述する。(その際の Bean 名などはコーディングポイントの項目を参照)

拡張処理が行われるタイミングは、下記シーケンス図の 3.の部分となる。

➤ シーケンス図(`getDelegateAction` メソッド)



➤ 解説(`getDelegateAction` メソッド)

`DelegatingRequestProcessorEx# getDelegateAction` メソッドの処理

1. 先に親クラスの `getDelegateAction` メソッドを実行する。
2. 拡張された `DelegateActionHandler` が存在した場合は、`DelegatingRequestProcessorEx` クラスは 1. での親クラスの結果に関わらず、`DelegateActionHandler` クラスに処理を委譲する。
3. `DelegateActionHandler` クラスは拡張された `getDelegateAction` メソッドを実行し、`Action` を返却する。

※1.と 3.の両方の処理で `action` が解決できた場合は、3. の拡張処理で取得した `action` が優先される。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.struts.action.RequestProcessorEx	TERASOLUNA Server Framework for Java (Web 版)が提供する RequestProcessor 拡張クラス。
2	jp.terasoluna.fw.struts.action.DelegatingRequestProcessorEx	RequestProcessorEx クラスを拡張し、5 つのメソッドについてハンドラへの処理の委譲を可能にしたクラス。ハンドラが用意されていない場合は、自動的に従来の処理を行う。
3	jp.terasoluna.fw.web.struts.action.handler.ProcessMultipartHandler	processMultipart 処理を拡張する際に利用するインタフェース。拡張処理の実装時はこのインタフェースを実現し、processMultipart メソッドをオーバーライドして、拡張処理を実装する。
4	jp.terasoluna.fw.web.struts.action.handler.ProcessActionFormHandler	processActionForm 処理を拡張する際に利用するインタフェース。拡張処理の実装時はこのインタフェースを実現し、processActionForm メソッドをオーバーライドして、拡張処理を実装する。
5	jp.terasoluna.fw.web.struts.action.handler.ProcessPopulateHandler	processPopulate 処理を拡張する際に利用するインタフェース。拡張処理の実装時はこのインタフェースを実現し、processPopulate メソッドをオーバーライドして、拡張処理を実装する。
6	jp.terasoluna.fw.web.struts.action.handler.ProcessValidateHandler	processValidate 処理を拡張する際に利用するインタフェース。拡張処理の実装時はこのインタフェースを実現し、processValidate メソッドをオーバーライドして、拡張処理を実装する。
7	jp.terasoluna.fw.web.struts.action.handler.DelegateActionHandler	getDelegateAction 処理を拡張する際に利用するインタフェース。拡張処理の実装時はこのインタフェースを実現し、getDelegateAction メソッドをオーバーライドして、拡張処理を実装する。

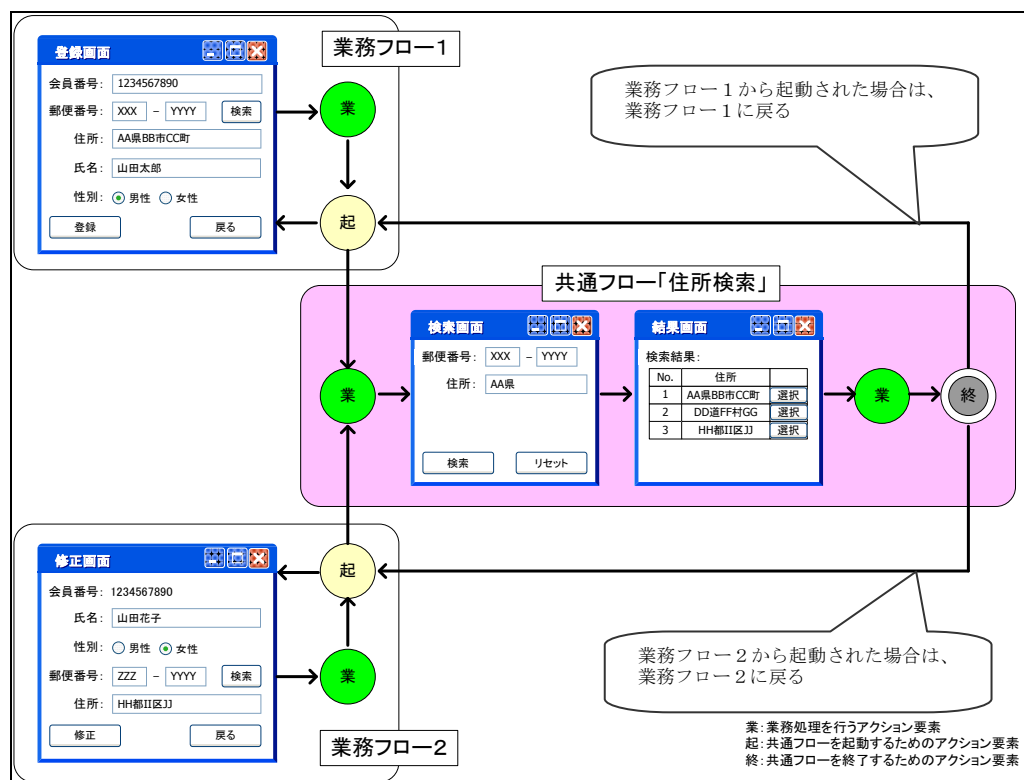
AL046 共通画面フロー機能

■ 概要

◆ 機能概要

- 複数の機能（アクション）から呼びだされることを前提とした一連の画面遷移（機能）を「共通画面フロー」と表現する。また、共通画面フローを呼び出す機能を、業務機能と呼ぶこととする。
- 共通画面フロー側（呼び出される側）のプログラムを変更することなく、呼び出し元の業務機能に戻るための、共通画面フロー起動専用アクション、および共通画面フロー終了専用アクションを提供する。

◆ 概念図



◆ 解説

- 複数の機能（アクション）から呼び出すことを前提とした一連の画面遷移を「共通画面フロー」として別機能に切り出す。
- 共通画面フローを呼び出す各業務機能側に、共通画面フローを起動するための専用のアクションを用意する。また共通画面フロー側では、自フローを終了し呼び出し元の業務機能（共通画面フローを起動したアクション）に戻るための専用のアクションを定義する。
- 共通画面フローの起動時および終了時にこれらのアクションを経由することで、共通画面フロー側のプログラムを変更せずに、呼び出し元の業務機能に戻ることができる。
- 共通画面フローの起動と終了のしくみ
 - 「共通画面フロー起動専用アクション」を実行すると、Struts 設定ファイルに定義した行き先の共通画面フロー名（後述の `destinationFlow` の値）をキーに、共通画面フロー起動専用アクションのアクションマッピング情報がセッションに登録される。
 - 「共通画面フロー終了専用アクション」は、共通画面フロー名（後述の `currentFlow` の値）を使って、共通画面フロー起動時に実行されたアクションのアクションマッピング情報をセッションから取得し、取得したアクションマッピング情報と `ActionServlet` のサーブレットマッピング情報から、共通画面フロー起動時に実行されたアクションを実行するためのパスを演算する。その後、「終点（出口）」名（後述の `terminal` の値）をリクエスト属性にセットし、「共通画面フロー起動専用アクション」にフォワードする。
 - 「共通画面フロー終了専用アクション」からフォワードで実行された「共通画面フロー起動専用アクション」は、「終点（出口）」名をリクエスト属性から取得し、「終点（出口）」名と一致する `name` 属性を持つ `forward` 要素を検出してフォワードする。

■ 使用方法

◆ コーディングポイント

業務機能側、共通画面フロー側の Struts 設定ファイル、および Bean 定義ファイルにそれぞれ、「共通画面フロー起動専用アクション」「共通画面フロー終了専用アクション」を定義する。共通画面フローの起動時には必ず「共通画面フロー起動専用アクション」を、終了時には「共通画面フロー終了専用アクション」を経由する。

以下、簡略化のため、共通画面フロー終了アクションから説明する。

- 共通画面フロー終了専用アクションの定義

共通画面フロー側の **Struts** 設定ファイル、および **Bean** 定義ファイルに、共通画面フローの「終点（出口）」に対応した、共通画面フロー終了専用アクションを定義する（複数の終点がある場合、その数だけ用意する）。

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

```
<action path="/commonflow1/Final01"
  className="jp.terasoluna.fw.ex.web.struts.action.FlowTerminateActionMapping">
  <set-property property="currentFlow" value="commonflow1" />
  <set-property property="terminal" value="Final01" />
</action>
```

共通画面フロー終了専用アクションを **Struts** 設定ファイルに定義する際は、以下の3点に留意する。

- （ア）**className** 属性は、**FlowTerminateActionMapping** とする。このクラスにできない場合は、拡張ポイントの「別の **ActionMapping** クラスを使用したい場合の対応方法」を参照のこと。
- （イ）**property** 属性が「**currentFlow**」固定、**value** 属性が共通画面フロー（の機能）名となる **set-property** 要素を1つ定義する。アプリケーション内に共通画面フローが複数存在する場合、共通画面フロー名はアプリケーション内で競合してはならない。
- （ウ）**property** 属性が「**terminal**」固定、**value** 属性が共通画面フローの「終点（出口）」名となる **set-property** 要素を1つ定義する。

➤ **Bean** 定義ファイル

```
<bean name="/commonflow1/Final01"
  class="jp.terasoluna.fw.ex.web.struts.actions.FlowTerminateAction"/>
```

共通画面フロー終了専用アクションを **Bean** 定義ファイルに定義する場合、**class** 属性に **FlowTerminateAction** を指定するだけでよい。

- 共通画面フロー起動専用アクションの定義

共通画面フローを呼び出す各業務機能側の **Struts** 設定ファイル、および **Bean** 定義ファイルに、共通画面フロー起動専用アクションを定義する。

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

```
<action path="/businessflow1/InvokeCommonFlow1"
  parameter="/commonflow1/Initialize.do"
  className="jp.terasoluna.fw.ex.web.struts.action.FlowInvokeActionMapping">
  <set-property property="destinationFlow" value="commonflow1" />
  <forward name="Final01" path="/businessflow1/Screen01.jsp" />
</action>
```

もしくは、

```
<action path="/businessflow1/InvokeCommonFlow1"
  className="jp.terasoluna.fw.ex.web.struts.action.FlowInvokeActionMapping">
  <set-property property="destinationFlow" value="commonflow1" />
  <forward name="success" path="/commonflow1/Initialize.do"/>
  <forward name="Final01" path="/businessflow1/Screen01.jsp" />
</action>
```

共通画面フロー起動専用アクションを Struts 設定ファイルに定義する際の留意点は以下 4 点である。

- (ア) **parameter** 属性、もしくは、**name** 属性値が「success」の **forward** 要素の **path** 属性にて、共通画面フロー内に定義されたアクションのパスを指定する。
(**FlowInvokeAction** からの共通画面フローへの遷移は、TERASOLUNA フレームワークの **ForwardAction** の機能を使用しているため、**ForwardAction** で遷移するときと同じ要領で記述する。モジュール分割を行っているアプリケーションでは、**forward** 要素を用いた方が記述しやすい。また、**parameter** 属性の指定方法は、**system.properties** の **forwardAction.contextRelative** の設定値によって、モジュール相対かコンテキスト相対かが変わる。)
- (イ) **className** 属性は **FlowInvokeActionMapping** とする。このクラスにできない場合は、拡張ポイントの「別の **ActionMapping** クラスを使用したい場合の対応方法」を参照のこと。
- (ウ) **property** 属性が「**destinationFlow**」固定、**value** 属性に行き先の共通画面フロー（の機能）名を指定した **set-property** 要素を 1 つ定義する。
- (エ) **name** 属性に呼び出す共通画面フローの終点（出口）名、**path** 属性にその終点（出口）から出た際の遷移先を記述した **forward** 要素を、必要に応じて 1 つ以上定義する。モジュール分割を行っている場合、業務機能側のモジュールのこのアクションから遷移するものとして記述する。例えば、**name** 属性値が「success」の **forward** 要素で **module** 属性を使用し、別モジュールに存在する共通画面フローを利用する場合であっても、共通画面フロー終了後の遷移先が呼び出し元の業務機能側のモジュールに存在する場合は、**name** 属性値が「Final01」等の **forward** 要素で **module** 属性を記述する必要はない。

➤ Bean 定義ファイル

```
<bean name="/businessflow1/InvokeCommonFlow1"
  class="jp.terasoluna.fw.ex.web.struts.actions.FlowInvokeAction"/>
```

共通画面フロー起動専用アクションを Bean 定義ファイルに定義する場合、**class** 属性に **FlowInvokeAction** を指定するだけでよい。

◆ 拡張ポイント

- 別の ActionMapping クラスを使用したい場合の対応方法

ActionMapping クラスとして、TERASOLUNA フレームワークの ActionMappingEx 以外のクラス(ActionMappingEx の拡張クラス)を使用しているアプリケーションで、この機能を使用する場合、コーディングポイントで指定した ActionMapping クラス (FlowInvokeActionMapping, FlowTerminateActionMapping) をそのまま使用することができない。

この場合、FlowInvokeActionMapping クラスの代わりとなる FlowInvokeActionMappingExtendPropertyHolder インタフェース実装クラスや、FlowTerminateActionMapping クラスの代わりとなる FlowTerminateActionMappingExtendPropertyHolder インタフェース実装クラスを用意することで、ActionMapping クラスの競合を回避できる。

(ただし、ActionMapping クラスに追加している属性名「destinationFlow」「currentFlow」「terminal」が競合してしまう場合、この機能を使うことはできない。)

状況に応じて、以下のいずれかの方法で対応する。

- アプリケーション用に拡張してある ActionMappingEx 拡張クラスに、FlowInvokeActionMappingExtendPropertyHolder インタフェースや FlowTerminateActionMappingExtendPropertyHolder インタフェースを実装する。この方法は、ActionMappingEx 拡張クラス自体を修正できる（ソースを管理している、あるいはこれから作る）ことが前提となる。
- アプリケーションで利用している ActionMappingEx 拡張クラスをさらに拡張し、FlowInvokeActionMappingExtendPropertyHolder インタフェースや FlowTerminateActionMappingExtendPropertyHolder インタフェースを実装する。この方法は、ActionMappingEx 拡張クラス自体が変更できなくとも適用可能な方法である。ただし、ActionMappingEx 拡張クラスが final クラスである場合には適用できない。

なお、いずれの方法においても、FlowInvokeActionMappingExtendPropertyHolder インタフェース実装クラスと FlowTerminateActionMappingExtendPropertyHolder インタフェース実装クラスは、別々のクラスとして用意しても、両インタフェースを実装した1つのクラスとして用意してもよい。また、インタフェースの実装方法は、FlowInvokeActionMapping クラスや FlowTerminateActionMapping クラスのソースを参照のこと。

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.ex.web.struts.action.DefaultFlowCallbackInfo	共通画面フロー終了時のコールバックに必要な情報を提供する、FlowCallbackInfo のデフォルト実装クラス。
2	jp.terasoluna.fw.ex.web.struts.action.DefaultFlowCallbackInfoFactory	共通画面フロー終了時のコールバックに必要な情報を提供する、DefaultFlowCallbackInfo のファクトリクラス。
3	jp.terasoluna.fw.ex.web.struts.action.FlowCallbackInfo	共通画面フロー終了時に、共通画面フロー起動アクションへコールバックするのに必要な情報を提供する処理を規定したインタフェース。
4	jp.terasoluna.fw.ex.web.struts.action.FlowCallbackInfoFactory	共通画面フロー終了時に利用する FlowCallbackInfo のファクトリを表すインタフェース。
5	jp.terasoluna.fw.ex.web.struts.action.FlowCallbackInfoMapping	共通画面フロー名と共通画面フロー終了時に利用する FlowCallbackInfo のマッピングを保持するクラス。
6	jp.terasoluna.fw.ex.web.struts.action.FlowInvokeActionMapping	共通画面フロー起動アクション専用の ActionMapping クラス。
7	jp.terasoluna.fw.ex.web.struts.action.FlowInvokeActionMappingExtendPropertyHolder	共通画面フロー起動アクション用の ActionMapping クラスが実装すべきインタフェース。
8	jp.terasoluna.fw.ex.web.struts.action.FlowTerminateActionMapping	共通画面フロー終了アクション専用の ActionMapping クラス。
9	jp.terasoluna.fw.ex.web.struts.action.FlowTerminateActionMappingExtendPropertyHolder	共通画面フロー終了アクション用の ActionMapping クラスが実装すべきインタフェース。
10	jp.terasoluna.fw.ex.web.struts.actions.FlowInvokeAction	共通画面フロー起動専用のアクションクラス。
11	jp.terasoluna.fw.ex.web.struts.actions.FlowTerminateAction	共通画面フロー終了専用のアクションクラス。
12	jp.terasoluna.fw.ex.web.struts.actions.FlowTerminateFailureException	共通画面フロー終了時の遷移先が見つからず、共通画面フローが正しく終了できないことを通知するための例外クラス。

■ 備考

- 共通画面フローの多重起動について
共通画面フロー機能では、共通画面フロー終了時にフォワードする際に必要な情報を、共通画面フロー名をキーにして管理している。そのため、複数の共通画面フローをネストさせることは可能だが、再起呼び出しとなるような呼び出しはできない。言い換えると、複数の異なる共通画面フローの多重起動は可能だが、同一の共通画面フローを多重起動することはできない。

例えば、業務機能から共通画面フローA を呼び出し、共通画面フローA から共通画面フローB を呼び出した場合は、共通画面フローB 終了時に共通画面フローA に戻り、共通画面フローA 終了時に業務機能に戻ることができる。共通画面フローB 終了後、共通画面フローA 終了前に、再度共通画面フローB を呼び出した場合も同様に、共通画面フローB、共通画面フローA が終了後、業務機能に戻ることができる。

一方、共通画面フローA から共通画面フローA を直接、または間接的に呼び出した場合、仮に共通画面フローの無限呼び出し防止のための分岐フローを用意したとしても、共通画面フローA が2度目に起動された時点で、1度目に起動された共通画面フローA から業務機能に戻るための情報が失われてしまうため、このような多重起動はできない。

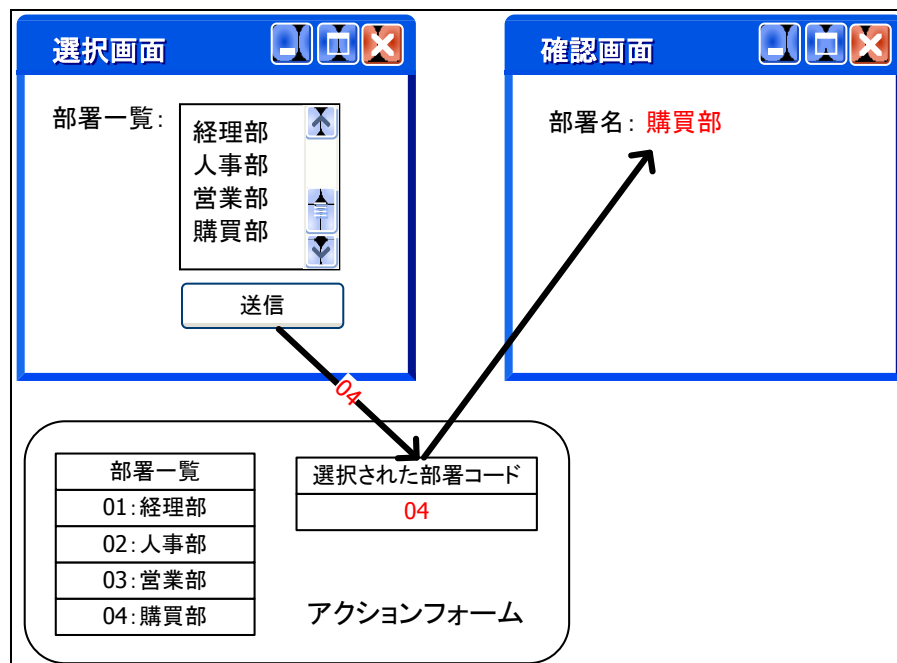
AL047 動的コードリスト値抽出・表示機能

■ 概要

◆ 機能概要

- 動的コードリストとは、ページスコープやリクエストスコープ、セッションスコープ、アクションフォーム内に保存された同型のオブジェクトの集合を表す（例えば「部署の一覧」など）。通常、セレクトボックスやチェックボックス、ラジオボタン等の限られた選択肢から値を選択する場合に用いる。
- 動的コードリストが TERASOLUNA フレームワークのコードリストと大きく異なる点は、ユーザごとに異なる内容のコードリストを動的に生成できる点である。
- 動的コードリスト内の各オブジェクトは、キーに該当する属性（「部署コード」など、以下キー属性と表記）と、画面に表示する属性（「部署名」など、以下表示属性と表記）を保持する。フォームサブミット時、選択されたオブジェクトのキー属性値（「部署コード：04」など）がアプリケーションサーバに送信される。
- 動的コードリスト値抽出・表示機能は、スコープから動的コードリストを検出し、指定されたキー値を持つオブジェクトを抽出して定義する、もしくは定義せずにそのオブジェクトの表示属性値を直接出力するカスタムタグを提供する（例えば、部署一覧から部署コード「04」に該当する「購買部」という値を出力する）。該当する値が見つからない場合、何も出力しない。

◆ 概念図



■ 使用方法

◆ コーディングポイント

- 動的コードリストの要素となるクラスの手配

動的コードリストの要素となるクラスを用意する。当該クラスは、キー属性と表示属性を持つ必要がある。表示属性は複数持つこともできる。以下は、キー属性として部署コード（deptCode）、表示属性として部署名（deptName）を持つ部署（Dept）クラスの例である。

➤ 部署（Dept）クラス

Dept
-deptCode : String -deptName : String
+getDeptCode() : String +getDeptName() : String

- 動的コードリストの生成

動的コードリストの要素となるクラスのインスタンスを生成してコレクション（ArrayList 等）や配列に格納し、ページスコープやリクエストスコープ、セッションスコープ、アクションフォームに登録する。

ここでは、下表に示した部署インスタンスのリスト（以下、部署リストと表記する）がアクションフォームクラス「SampleForm」の「deptList」というプロパティに設定されており、アクションフォームは「_SampleForm」という名前でセッションスコープに登録されているものとする。

また、セレクトボックスやラジオボタン等で選択された部署コードが、SampleForm の selectedDeptCode という文字列型のフィールドに保持されていると想定する。

➤ 部署リスト（deptList）

部署コード	部署名
01	経理部
02	人事部
03	営業部
04	購買部

➤ アクションフォーム（SampleForm）クラス

SampleForm
-deptList : List<Dept> -selectedDeptCode : String
+getDeptList() : List<Dept> +getSelectedDeptCode() : String +setSelectedDeptCode(String selectedDeptCode) : void

- extract タグの使用

JSP に `extract` カスタムタグを記述し、指定されたキー値を持つ動的コードリスト内のオブジェクトの表示属性を画面に出力する。下表に、`extract` タグがサポートする属性の一覧を示す。

➤ `extract` タグがサポートする属性の一覧

属性名	デフォルト	必須	概要
<code>collection</code>	-	△	動的コードリストを直接指定する。 <code>collectionName</code> 属性を設定していない場合は、必ず設定しなければならない。 例) <code>\${_SampleForm.deptList}</code>
<code>collectionName</code>	-	△	動的コードリストを保持する Bean がリクエストスコープやセッションスコープに登録されている属性名を指定する。 <code>collection</code> 属性を設定していない場合は、必ず設定しなければならない。 例) <code>_SampleForm</code>
<code>collectionProperty</code>	-	-	<code>collectionName</code> 属性と組み合わせて使用する。 <code>collectionName</code> 属性で指定された Bean の中のある属性が動的コードリストである場合に、その属性名を指定する。省略時は <code>collectionName</code> 属性で指定された Bean 自体が動的コードリストであるとみなす。 例) <code>deptList</code>
<code>collectionScope</code>	-	-	<code>collectionName</code> 属性と組み合わせて使用する。動的コードリストを検索する範囲を、 <code>page</code> 、 <code>request</code> 、 <code>session</code> 、 <code>application</code> のいずれかから指定する。省略時は <code>page</code> 、 <code>request</code> 、 <code>session</code> 、 <code>application</code> の順に自動的に検索される。
<code>keyProperty</code>	-	○	動的コードリスト内の要素の、キー属性を指定する。 例) <code>deptCode</code>

これらの属性は、動的コードリストを検出し、キー属性を特定するために使用する。`collection` 属性を使用するか、もしくは `collectionName` 属性、`collectionProperty` 属性、`collectionScope` 属性の組み合わせのいずれかの方法で動的コードリストを指定する必要がある。なお、`collection` 属性を指定した場合には、その値が `null` である場合を除き、`collectionName` 属性、`collectionProperty` 属性、`collectionScope` 属性は無視される。

動的コードリストとして、`null` を設定することはできない。

➤ extract タグがサポートする属性の一覧（続き）

属性名	デフォルト	必須	概要
condition	-	△	動的コードリストから値を取得するための抽出条件を直接指定する。conditionName 属性を設定していない場合は、必ず設定しなければならない。 例) \${_SampleForm.selectedDeptCode}
conditionName	-	△	動的コードリストから値を取得するための抽出条件を保持する Bean が、リクエストスコープやセッションスコープに登録されている属性名を指定する。condition 属性を設定していない場合は、必ず設定しなければならない。 例) _SampleForm
conditionProperty	-	-	conditionName 属性と組み合わせて使用する。conditionName 属性で指定された Bean の中のある属性が抽出条件である場合に、その属性名を指定する。省略時は conditionName 属性で指定された Bean 自体が抽出条件であるとみなす。 例) selectedDeptCode
conditionScope	-	-	conditionName 属性と組み合わせて使用する。動的コードリストから値を取得するための抽出条件を保持する Bean を検索する範囲を、page、request、session、application のいずれかから指定する。省略時は page、request、session、application の順に自動で検索される。

上記の属性は、動的コードリストから値を取得するための抽出条件を特定するために使用する。condition 属性を使用するか、もしくは conditionName 属性、conditionProperty 属性、conditionScope 属性の組み合わせのいずれかの方法で抽出条件を指定する必要がある。なお、condition 属性を指定した場合には、その値が null である場合を除き、conditionName 属性、conditionProperty 属性、conditionScope 属性は無視される。

抽出条件には、単一のキー値か、キー値のコレクションや配列を設定できる。

抽出条件として、null を設定することはできない。ユーザの操作により（特に、チェックボックスを1つも選択しなかったとき等に）抽出条件が null となる場合がある場合には、このタグを使用する前に、logic:present タグ等で null チェックを行う必要がある。

➤ extract タグがサポートする属性の一覧（続き）

属性名	デフォルト	必須	概要
id	-	△	動的コードリストから抽出した要素を Bean として定義する際の名前を設定する。動的コードリストから要素を抽出し、その抽出結果を Bean として定義する際に使用する。 writeProperty 属性を設定していない場合は、必ず設定しなければならない。また、 writeProperty 属性と併用してはならない。 抽出条件がキー値のコレクションや配列である場合には、複数の抽出結果のレンダリングを JSP で実装する必要があるため、この属性を使用しなければならない。 例) selectedDept
toScope	-	-	id 属性と組み合わせて使用する。 Bean を定義する先のスコープを、 page 、 request 、 session 、 application のいずれかから指定する。省略時は page スコープに定義される。
writeProperty	-	△	動的コードリスト内の要素の、表示属性を指定する。この属性を使用すると、表示属性の表示を行う。 id 属性を設定していない場合は、必ず設定しなければならない。また、 id 属性と併用してはならない。 抽出条件がキー値のコレクションや配列である場合には、複数の抽出結果のレンダリングを JSP で実装する必要があるため、この属性ではなく、 id 属性を使用しなければならない。 例) deptName
filter	true	-	writeProperty 属性と組み合わせて使用する。動的コードリスト内の要素から取得した表示属性値を HTML エスケープするか否かを設定する。 HTML エスケープしたくない場合のみ、 false に設定する。

上記の属性は、このタグによる抽出結果を定義するか、定義せずに直接表示するか、直接表示するのであればどの属性を表示するかを特定するために使用する。**id** 属性を使用した場合は抽出結果の定義（**bean:write** や **logic:iterate** 等の **name** 属性で参照可能になる）、**writeProperty** 属性を使用した場合は抽出結果の直接表示という動作になる。

writeProperty 属性を使用した際に表示されるのは、動的コードリスト内の要素の表示属性の値だが、**id** 属性を使用した際に抽出結果として定義されるのは、表示属性の値ではなく、動的コードリスト内の要素、あるいは、そのコレクションである。

抽出結果として定義されるのが、単一の要素となるか、要素のコレクションとなるかは、抽出条件が単一のキー値であるか、キー値のコレクションや配列であるかで決まる。

- ・ 抽出条件が単一のキー値である場合、動的コードリスト内の該当要素 1 つを抽出結果として定義する。
- ・ 抽出条件がキー値のコレクションや配列である場合、動的コードリスト内の該当要素を格納するコレクションを抽出結果として定義する。
 - 抽出条件のコレクションや配列の要素数が 0 件である場合、0 件のコレクションを抽出結果として定義する。

● extract タグの使用例（単一選択、単一項目表示）

extract タグを使用し、_SampleForm 内の selectedDeptCode 属性に保持されている部署コードに対応する部署名を出力する JSP の記述例を挙げる。なお、以下 4 例はいずれも同じ結果を出力する。

➤ JSP 記述例（collection 属性を指定する場合）

```
<tl:extract
  collection="{$_SampleForm.deptList}"
  keyProperty="deptCode"
  conditionName="_SampleForm" conditionProperty="selectedDeptCode"
  writeProperty="deptName" />
```

➤ JSP 記述例（collectionName 属性と collectionProperty 属性を指定する場合）

```
<tl:extract
  collectionName="_SampleForm" collectionProperty="deptList"
  keyProperty="deptCode"
  conditionName="_SampleForm" conditionProperty="selectedDeptCode"
  writeProperty="deptName" />
```

➤ JSP 記述例（condition 属性を指定する場合）

```
< tl:extract
  collection="{$_SampleForm.deptList}"
  keyProperty="deptCode"
  condition="{$_SampleForm.selectedDeptCode}"
  writeProperty="deptName" />
```

➤ JSP 記述例 (id 属性で定義してから bean:write で出力する場合)

```
<tl:extract
  collection="${_SampleForm.deptList}"
  keyProperty="deptCode"
  conditionName="_SampleForm" conditionProperty="selectedDeptCode"
  id="selectedDept" />
<bean:write
  name="selectedDept" property="deptName" ignore="true" />
```

● extract タグの使用例 (単一選択、複数項目表示)

extract タグを使用し、_SampleForm 内の selectedDeptCode 属性に保持されている部署コードに対応する部署名と略称を出力する JSP の記述例を挙げる。なお、extract タグの id 属性と bean:write タグを組み合わせる表示する方法と、writeProperty 属性を使って直接表示する方法があるが、このケースでは、writeProperty 属性を使って直接表示する方法は性能アンチパターンとなるため、ここでは記載しない。

※この例では、前述の「部署 (Dept) クラス」に、表示属性として略称 (deptShortName) が追加で定義されているものとする。

➤ JSP 記述例

```
<tl:extract
  collection="${_SampleForm.deptList}"
  keyProperty="deptCode"
  conditionName="_SampleForm" conditionProperty="selectedDeptCode"
  id="selectedDept" />
<bean:write
  name="selectedDept" property="deptName" ignore="true" /> ←部署名
<bean:write
  name="selectedDept" property="deptShortName" ignore="true" /> ←略称
(<td>や<br>等、レンダリング用の HTML タグは記載を省略している。)
```

- extract タグの使用例（複数選択、単一・複数項目表示）

extract タグを使用し、_SampleForm 内の selectedDeptCodes 属性に保持されている部署コード配列に対応する部署名リストを出力する JSP の記述例を挙げる。なお、extract タグの id 属性と bean:write タグを組み合わせる表示する方法と、writeProperty 属性を使って直接表示する方法があるが、このケースでは、writeProperty 属性を使って直接表示する方法は性能アンチパターンとなるため、ここでは記載しない。

※この例では、前述の「アクションフォーム（SampleForm）クラス」の、String 型の selectedDeptCode 属性とその getter/setter に代わり、String 配列型の selectedDeptCodes 属性とその getter/setter が定義されているものとする。

➤ JSP 記述例

```
<tl:extract
  collection="${_SampleForm.deptList}"
  keyProperty="deptCode"
  conditionName="_SampleForm" conditionProperty="selectedDeptCodes"
  id="selectedDeptList" />
<logic:iterate name="selectedDeptList" id="selectedDept">
  <tr>
    <td><bean:write name="selectedDept" property="deptName" /> </td>
    （部署名以外の項目も表示したい場合は、ここに bean:write タグを追加する。）
  </tr>
</logic:iterate>
```

◆ 拡張ポイント

なし

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.ex.web.taglib.ExtractTag	動的コードリストから、指定されたキー値を持つオブジェクトを抽出するカスタムタグ。
2	jp.terasoluna.fw.ex.web.taglib.ExtractTagExtraInfo	extract タグのための TagExtraInfo クラス。

■ 備考

- 性能上の留意点

`extract` タグの実行時には、動的コードリスト内の順検索が行われる。性能的な観点で、無駄な検索（同一動的コードリスト、同一キー属性、同一抽出条件での検索）が何度も行われることのないよう、`id` 属性を適宜使用し、このタグの実行回数を最小限にとどめること。

- ・ 表示属性が複数項目ある場合は、`writeProperty` 属性値だけ異なる `extract` タグを複数配置するのではなく、`extract` タグの `id` 属性で一旦定義してから、`property` 属性だけ異なる `bean:write` タグを複数配置する。（`extract` タグの `id` 属性で与えた名前を `bean:write` タグの `name` 属性に設定し、表示属性の名前を `bean:write` タグの `property` 属性に設定する。）

- コーディングポイントの「`extract` タグの使用例（単一選択、複数項目表示）」を参照

- ・ チェックボックスで複数選択された結果を表示する場合等、抽出条件となるキー値が複数存在しうる場合は、キー値のコレクションでイテレートする `logic:iterate` タグの中で `extract` タグを使用するのではなく、`extract` タグの抽出条件にキー値のコレクションを与え、その抽出結果のコレクションでイテレートする `logic:iterate` タグの中で `bean:write` タグを使用する。（`extract` タグの `id` 属性で与えた名前を `logic:iterate` タグの `name` 属性に設定し、`logic:iterate` タグの `id` 属性で与えた名前を `bean:write` タグの `name` 属性に設定し、表示属性の名前を `bean:write` タグの `property` 属性に設定する。）

- コーディングポイントの「`extract` タグの使用例（複数選択、単一・複数項目表示）」を参照

- キー属性と抽出条件の型について

キー属性と抽出条件の型は合わせる必要がある。通常、抽出条件はアクションフォームのプロパティであり、その型は、リクエストパラメータからの型変換が不要な `String` や `String` 配列とするため、この場合、動的コードリスト内の要素のキー属性（例では `Dept` クラスの `deptCode`）の型も `String` とする。

- カスタムタグの使用準備
カスタムタグを使用する設定が未設定の場合、以下の設定を行う。

- terasoluna-taglibex-x.x.x.jar を WEB-INF/lib に配置する。
- web.xml に、以下の taglib 要素を追加する。

```
<taglib>  
  <taglib-uri>/terasoluna-library</taglib-uri>  
  <taglib-location>/WEB-INF/tlds/terasoluna-library.tld</taglib-location>  
</taglib>
```

- web.xml の taglib-location で指定した場所に、terasoluna-library.tld を配置する。
terasoluna-library.tld のファイルは、terasoluna-taglibex-x.x.x.jar の中の、META-INF/tld 内に存在する。
- JSP に、以下の taglib ディレクティブを追加する。

```
<%@ taglib uri="/terasoluna-library" prefix="tl"%>
```

AL055 拡張ページリンク機能

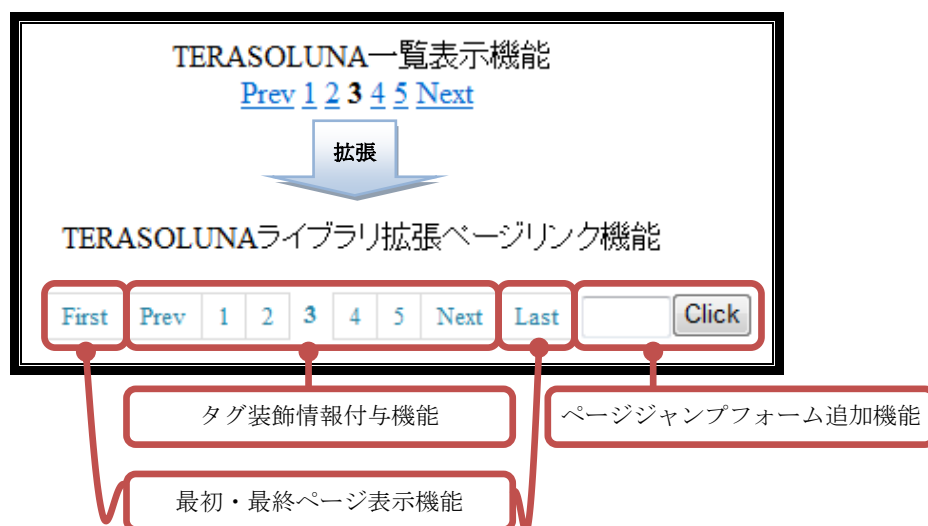
■ 概要

◆ 機能概要

TERASOLUNA が提供する<ts:pageLinks>要素を拡張した機能である。<ts:pageLinks>の詳細については「WI-01 一覧表示機能」を参照のこと。

- 以下の拡張機能を利用する事ができる。
 - 「最初・最後ページリンク機能」を利用する事が出来る。
 - 「ページジャンプフォーム追加機能」を利用する事出来る。
 - 「タグ装飾情報付与機能」を利用することが出来る。

◆ 概念図



■ 使用方法

◆ コーディングポイント

<ts:pageLinks>に準ずる。以下では、<ts:pageLinks>との差分を記載する。

- 最初・最後ページリンク機能

プロパティファイル(pageLinks.properties)に出力するリンクのフォーマットを追加することで最初のページ、最後のページへのリンクを出力できる。以下に設定対象のプロパティを記述する。

キー	値(デフォルトなし)	概要
pageLinks.first.char	(例) First	現在のページから最初のページに遷移するリンクの文字列を出力する。設定しない場合、リンクは非表示となる。
pageLinks.last.char	(例) Last	現在のページから最後のページに遷移するリンクの文字列を出力する。設定しない場合、リンクは非表示となる。

- ページジャンプフォーム機能

ts:pageLinks タグの pageJump 属性を true に指定することで、テキストフォームに入力したページ番号へ直接ジャンプすることが可能なページジャンプフォームを出力出来る。ページジャンプ用テキストフォームは JavaScript にて入力チェックを行っており、0 以下の数字および数字以外の文字列が入力された場合はページジャンプを行わない。ページジャンプフォーム機能は submit 属性が true の場合は使用できない。

- submit 属性が false (または未設定) の場合の使用例

- ◇ JSP

```
<tl:pageLinksEx action="/list" rowProperty="row" totalProperty="totalCount" indexProperty="startIndex"
  pageJump="true"/>
```

pagejump 属性を true に指定する。

以下に設定することができるプロパティを記述する。

キー	値	デフォルト	概要
pageLinks.jump.text.attributes	(例) size="2"	-	ページジャンプ用テキストフォームの属性設定フィールド。size や maxlength 等の属性を設定することが出来る。
pageLinks.jump.submit.attributes	(例) class="jump"	-	ページジャンプ用サブミットボタンの属性設定フィールド。class 等の属性を設定することが出来る。
pageLinks.jump.submit.char	(例) Click	submit	ページジャンプ用サブミットボタンの value 属性を設定することが出来る。設定しない場合、"submit" がデフォルト出力される。

- タグ装飾情報付与

プロパティファイル(pageLinks.properties)にタグ装飾情報を設定することで、出力するリンクの前後にタグ付与し、装飾することができる。以下に設定対象のプロパティを記述する。

キー	値(デフォルトなし)	概要
pageLinks.start.tag pageLinks.end.tag	(例) など	ページリンク表示部分の開始・終了タグ装飾情報を出力する。
pageLinks.start.page.tag pageLinks.end.page.tag	(例) など	各ページに遷移するリンク文字列表示部分の開始・終了タグ装飾情報を出力する。
pageLinks.start.prev.tag pageLinks.end. prev.tag	(例) <li class="prev"> など	前のページに遷移するリンク文字列表示部分の開始・終了タグ装飾情報を出力する。 設定しない場合、各ページに遷移するリンク文字列の開始・終了タグ装飾情報が代替される。
pageLinks.start.next.tag pageLinks.end. next.tag	(例) <li class="next"> など	次のページに遷移するリンク文字列表示部分の開始・終了タグ装飾情報を出力する。 設定しない場合、各ページに遷移するリンク文字列の開始・終了タグ装飾情報が代替される。
pageLinks.start.first.tag pageLinks.end. first.tag	(例) <li class="first"> など	最初のページに遷移するリンク文字列表示部分の開始・終了タグ装飾情報を出力する。設定しない場合、前のページに遷移するリンク文字列表示部分の開始・終了タグ装飾情報を、前のページに遷移するリンク文字列表示部分の開始・終了タグ装飾情報が設定されていない場合は各ページに遷移するリンク文字列の開始・終了タグ装飾情報が代替される
pageLinks.start.last.tag pageLinks.end. last.tag	(例) <li class="last"> など	最後のページに遷移するリンク文字列表示部分の開始・終了タグ装飾情報を出力する。設定しない場合、次のページに遷移するリンク文字列表示部分の開始・終了タグ装飾情報を、次のページに遷移するリンク文字列表示部分の開始・終了タグ情報が設定されていない場合は各ページに遷移するリンク文字列の開始・終了タグ装飾情報が代替される。
pageLinks.start.active.tag pageLinks.end. active.tag	(例) <li class="active"> など	現在ページ文字列表示部分の開始・終了タグ装飾情報を出力する。設定しない場合、各ページに遷移するリンク文字列の開始・終了タグ装飾情報が代替される。
pageLinks.start.disable.tag pageLinks.end. disable.tag	(例) <li class="disable"> など	リンクなし文字列表示部分の開始・終了タグ装飾情報を出力する。設定しない場合、各ページに遷移するリンク文字列の開始・終了タグ装飾情報が代替される。
pageLinks.start.jump.tag pageLinks.end. jump.tag	(例) <li class="jump"> など	ページジャンプフォーム表示部分の開始・終了タグ装飾情報を出力する。設定しない場合、各ページに遷移するリンク文字列の開始・終了タグ装飾情報が代替される。

➤ 使用例

◇ pageLinks.properties

WI-01 一覧表示機能用設定

```
pageLinks.maxDirectLinkCount=5
```

```
pageLinks.prev1.char=Prev
```

```
pageLinks.next1.char=Next
```

#最初・最後ページリンク追加用設定

```
pageLinks.first.char=First
```

```
pageLinks.last.char=Last
```

#タグ装飾情報付与機能用設定

```
pageLinks.start.tag=<ul>
```

```
pageLinks.end.tag=</ul>
```

```
pageLinks.start.page.tag=<li>
```

```
pageLinks.end.page.tag=</li>
```

```
pageLinks.start.first.tag=<li class="first">
```

```
pageLinks.end.first.tag=</li>
```

```
pageLinks.start.last.tag=<li class="last">
```

```
pageLinks.end.last.tag=</li>
```

```
pageLinks.start.prev.tag=<li class="prev">
```

```
pageLinks.end.prev.tag=</li>
```

```
pageLinks.start.next.tag=<li class="next">
```

```
pageLinks.end.next.tag=</li>
```

```
pageLinks.start.active.tag=<li class="active">
```

```
pageLinks.end.active.tag=</li>
```

```
pageLinks.start.disable.tag=<li class="disable">
```

```
pageLinks.end.disable.tag=</li>
```

```
pageLinks.start.jump.tag=<li class="jump">
```

```
pageLinks.end.jump.tag=</li>
```

#ページジャンプフォーム追加機能用設定

```
pageLinks.jump.text.attributes=size="2" maxlength="3"
```

```
pageLinks.jump.submit.attributes=class="DisableDoubleClick"
```

```
pageLinks.jump.submit.char=Click
```

◇ 画面出力例

```

<ul>
  <li class="first"><a href="/test/list.do?row=10&startIndex=0">First</a></li>
  <li class="disable">Prev</li>
  <li><a href="/test/list.do?row=10&startIndex=0">1</a></li>
  <li class="active">2</li>
  <li><a href="/test/list.do?row=10&startIndex=20">3</a></li>
  <li><a href="/test/list.do?row=10&startIndex=30">4</a></li>
  <li><a href="/test/list.do?row=10&startIndex=40">5</a></li>
  <li class="next"><a href="/test/list.do?row=10&startIndex=30">Next</a></li>
  <li class="last"><a href="/test/list.do?row=10&startIndex=40">Last</a></li>
  <li class="jump">
    <form name="_listForm" method="post" action="/test/list.do"
      onsubmit="return teralib.pageLinkJump (this, 10,'startIndex')"/>
      <input type="hidden" name="row" value="10"/>
      <input type="hidden" name="startIndex"/>
      <input type="text" name="jumpIndex" size="2" maxlength="3"/>
      <input type="submit" value="Click" class="DisableDoubleClick"/>
    </form>
  </li>
</ul>

```

● 拡張ページリンク使用手順

1. JSP

JSP ページのページリンクを出力する場所にタグを記述する。

➤ JSP

```
<tl:pageLinksEx action="/list" rowProperty="row" totalProperty="totalCount" indexProperty="startIndex" />
```

➤ JSP（ページジャンプフォーム機能を使用する場合）

```
<tl:pageLinksEx action="/list" rowProperty="row" totalProperty="totalCount" indexProperty="startIndex"
  pageJump="true"/>
```

pagejump 属性を true に指定する。

➤ JSP(サブミットを行いたい場合)

```
<ts:form action="/list" styleId="_listForm_pageLinks">
  <tl:pageLinksEx action="/list"
    rowProperty="row"
    totalProperty="totalCount"
    indexProperty="startIndex"
    submit="true" />
</ts:form>
```

サブミットを行いたい場合は、<ts:form>タグに styleId 属性で「アクションフォーム名_pageLinks」を指定する

submit 属性を true に指定する

2. 参考: Struts 設定ファイル設定例

一覧情報をアクションフォームから取得する場合の設定例を示す。一覧情報を常にデータベースから取得する使用例、その他詳細については「WI-01 一覧表示機能」を参照のこと。

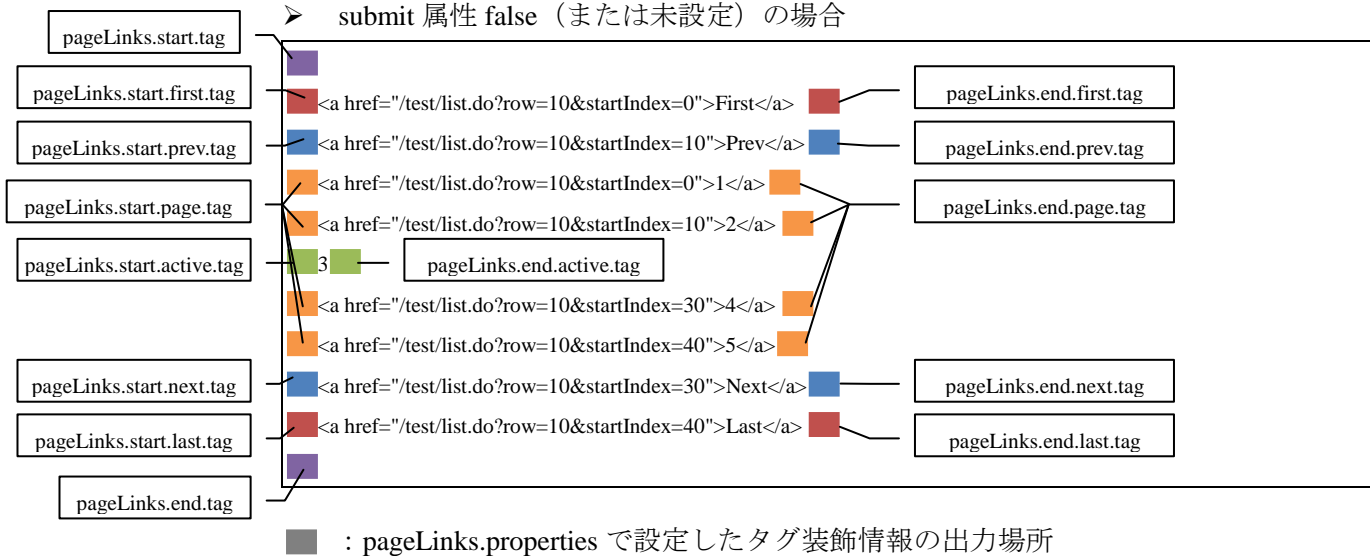
➤ Struts 設定ファイル

```
<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-property name="totalCount" type="java.lang.String" />
  </form-bean>
</form-beans>

<action-mappings type="jp.terasoluna.fw.web.struts.action.ActionMappingEx">
  <action path="/list" name="_listForm" scope="session">
    <forward name="success" path="/listSCR.do" />
  </action>
  <action path="/listSCR" name="_listForm" scope="session" parameter="/list.jsp"/>
</action-mappings>
```

3. 参考: 画面出力例

➤ submit 属性 false（または未設定）の場合

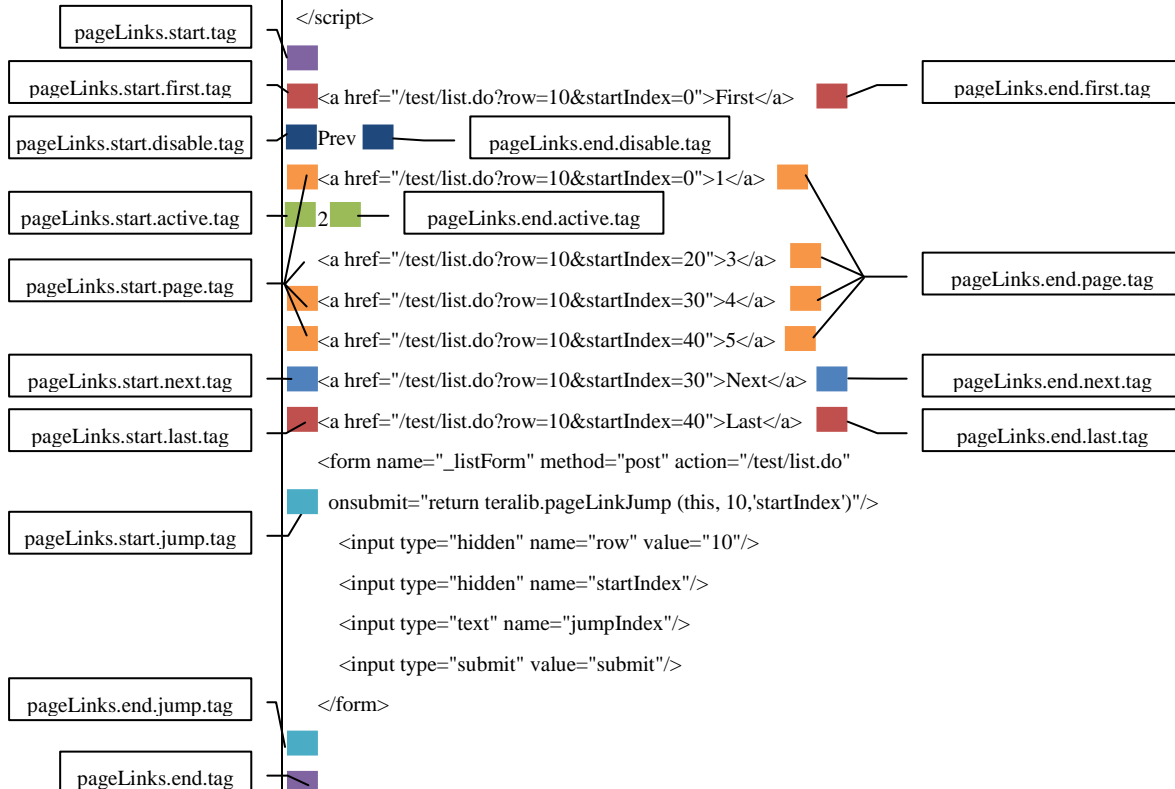


- submit 属性 false（または未設定）でページジャンプフォーム機能を使用する場合

```

<script type="text/javascript">
<!--
  if(typeof(teralib) === "undefined"){
    teralib = {};
  }
  teralib.pageLinkJump = function(form, row, indexProperty){
    var jump = form.elements['jumpIndex'];
    var index = form.elements[indexProperty];
    var idx;
    var jumpValue = parseInt(jump.value,10);
    if( jumpValue == 2 ){
      return false;
    } else if( 0 < jumpValue ){
      idx = (jumpValue -1) * 10;
      index.value = idx;
    } else {
      return false;
    }
  }
}
// -->
</script>
<a href="/test/list.do?row=10&startIndex=0">First</a>
Prev
<a href="/test/list.do?row=10&startIndex=0">1</a>
2
<a href="/test/list.do?row=10&startIndex=20">3</a>
<a href="/test/list.do?row=10&startIndex=30">4</a>
<a href="/test/list.do?row=10&startIndex=40">5</a>
<a href="/test/list.do?row=10&startIndex=30">Next</a>
<a href="/test/list.do?row=10&startIndex=40">Last</a>
<form name="_listForm" method="post" action="/test/list.do"
  onsubmit="return teralib.pageLinkJump (this, 10,'startIndex')"/>
  <input type="hidden" name="row" value="10"/>
  <input type="hidden" name="startIndex"/>
  <input type="text" name="jumpIndex"/>
  <input type="submit" value="submit"/>
</form>

```



➤ submit 属性 true の場合

```

<form name="_listForm" method="post" action="/test/list.do" id="_listForm_pageLinks">
  <input type="hidden" name="row" value="5"/>
  <input type="hidden" name="startIndex" value="5"/>

  <script type="text/javascript">
  <!--
    if(typeof(teralib) === "undefined"){
      teralib = {};
    }
    teralib.pageLinkSubmit = function(rowProperty, indexProperty, row, startIndex){
      var submitForm = document.getElementById('_listForm_pageLinks');
      submitForm[rowProperty].value = row;
      submitForm[indexProperty].value = startIndex;
      submitForm.submit();
    }
  // -->
  </script>
  <a href="#" onclick="return teralib.pageLinkSubmit('row','startIndex',0,0)">First</a>
  <a href="#" onclick="return teralib.pageLinkSubmit('row','startIndex',10,0)">Prev</a>
  <a href="#" onclick="return teralib.pageLinkSubmit('row','startIndex',0,0)">1</a>
  <a href="#" onclick="return teralib.pageLinkSubmit('row','startIndex',10,0)">2</a>
  3
  <a href="#" onclick="return teralib.pageLinkSubmit('row','startIndex',30,0)">4</a>
  <a href="#" onclick="return teralib.pageLinkSubmit('row','startIndex',40,0)">5</a>
  <a href="#" onclick="return teralib.pageLinkSubmit('row','startIndex',30,0)">Next</a>
  <a href="#" onclick="return teralib.pageLinkSubmit('row','startIndex',40,0)">Last</a>
</form>

```

pageLinks.start.tag

pageLinks.start.first.tag

pageLinks.start.prev.tag

pageLinks.start.page.tag

pageLinks.start.active.tag

pageLinks.start.next.tag

pageLinks.start.last.tag

pageLinks.end.tag

pageLinks.end.first.tag

pageLinks.end.prev.tag

pageLinks.end.page.tag

pageLinks.end.active.tag

pageLinks.end.next.tag

pageLinks.end.last.tag

■ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.struts.taglib.PageLinksTagEx	<ts:pageLinks>タグを拡張した、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

- 追加タグ属性(その他は<ts:pageLinks>のタグ属性一覧に準ずる)

属性	必須	デフォルト	概要
pageJump	-	false	拡張機能のページジャンプフォームを使用する場合は true を指定する。submit 属性=true 時は使用できない。

<ts:pageLinks>の詳細については「WI-01 一覧表示機能」を参照のこと。

■ 関連機能

「WI-01 一覧表示機能」を参照のこと。

■ 備考

- カスタムタグの使用準備
カスタムタグを使用する設定が未設定の場合、以下の設定を行う。

- terasoluna-taglibex-x.x.x.jar を WEB-INF/lib に配置する。
- web.xml に、以下の taglib 要素を追加する。

```
<taglib>
  <taglib-uri>/terasoluna-library</taglib-uri>
  <taglib-location>/WEB-INF/tlds/terasoluna-library.tld</taglib-location>
</taglib>
```

- web.xml の taglib-location で指定した場所に、terasoluna-library.tld を配置する。
terasoluna-library.tld のファイルは、terasoluna-taglibex-x.x.x.jar の中の、META-INF/tld 内に存在する。

- JSP に、以下の taglib ディレクティブを追加する。

```
<%@ taglib uri="/terasoluna-library" prefix="tl"%>
```