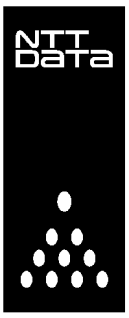


TERASOLUNA Server Framework for Java (Rich 版)

機能説明書

第 2.0.3.0 版

株式会社ＮＴＴデータ

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

(1)本ドキュメントの著作権及びその他一切の権利は、NTT データあるいは NTT データに権利を許諾する第三者に帰属します。

(2)本ドキュメントの一部または全部を、自らが使用する目的において、複製、翻訳、翻案することができます。ただし本ページの規約全文、および NTT データの著作権表示を削除することはできません。

(3)本ドキュメントの一部または全部を、自らが使用する目的において改変したり、本ドキュメントを用いた二次的著作物を作成することができます。ただし、「参考文献:TERASOLUNA Server Framework for Java (Rich 版) 機能説明書」あるいは同等の表現を、作成したドキュメント及びその複製物に記載するものとします。

(4)前2項によって作成したドキュメント及びその複製物を、無償の場合に限り、第三者へ提供することができます。

(5)NTT データの書面による承諾を得ることなく、本規約に定められる条件を超えて、本ドキュメント及びその複製物を使用したり、本規約上の権利の全部又は一部を第三者に譲渡したりすることはできません。

(6)NTT データは、本ドキュメントの内容の正確性、使用目的への適合性の保証、使用結果についての的確性や信頼性の保証、及び瑕疵担保義務も含め、直接、間接に被ったいかなる損害に対しても一切の責任を負いません。

(7)NTT データは、本ドキュメントが第三者の著作権、その他如何なる権利も侵害しないことを保証しません。また、著作権、その他の権利侵害を直接又は間接の原因としてなされる如何なる請求(第三者との間の紛争を理由になされる請求を含む。)に関しても、NTT データは一切の責任を負いません。

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

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

本書は、TERASOLUNA Server Framework for Java（Rich 版） ver2.0.3.0 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 (RG-01 DB メッセージ機能)	DB メッセージ機能を共通機能化し、機能名を変更
〃	〃	CB-01 データベースアクセス機能	QueryRowHandleDAO に関する記述を追加
〃	2009/09/30	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
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-messeges, application-messages に変更
〃	〃	CD-01 ユーティリティ機能 RB-04 制御情報管理機能	誤字を修正
〃	〃	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

目次

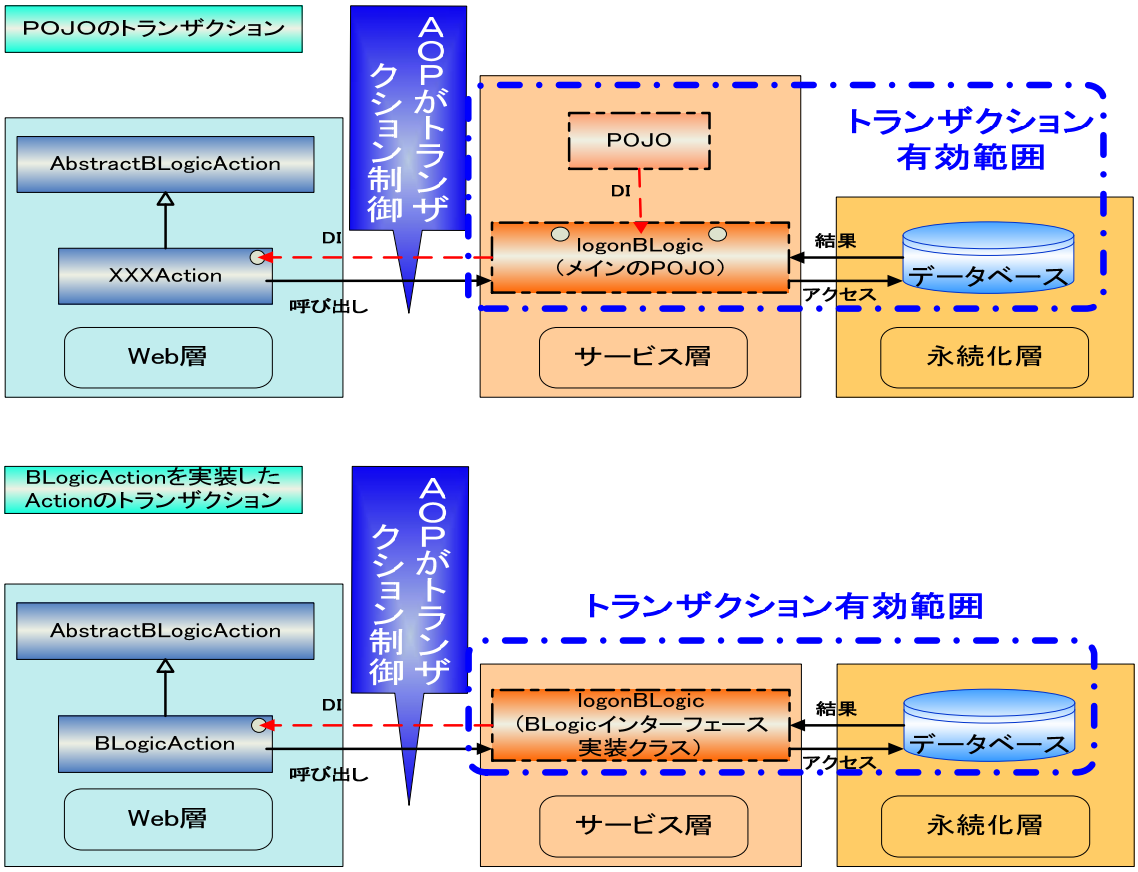
CA-01 トランザクション管理機能

■ 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。

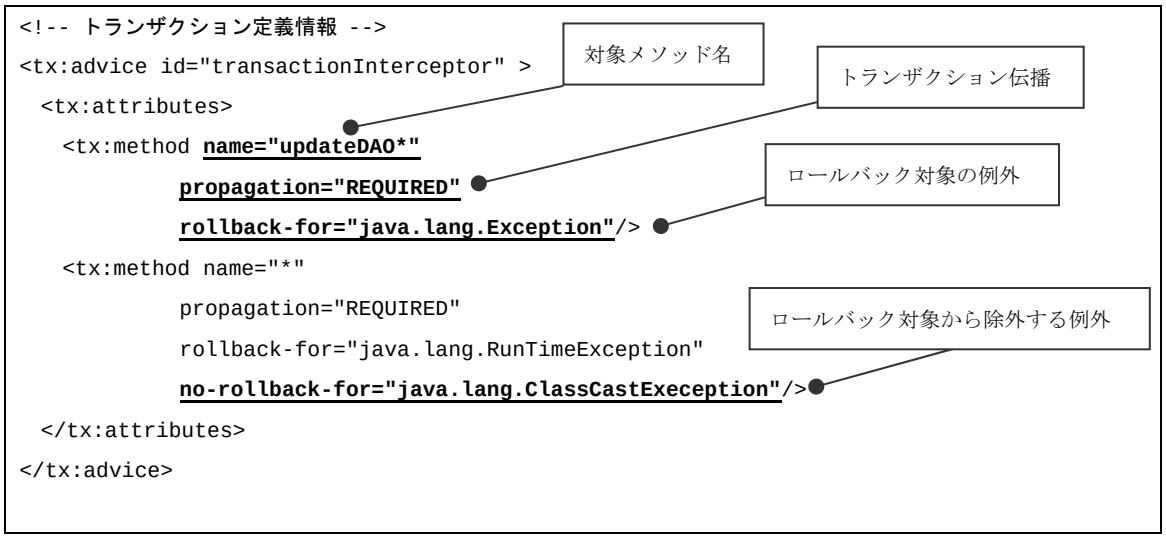
Spring では、その他にも独立性レベルやタイムアウト、読取専用などの定義情報の設定が可能である。詳細は、SpringAPI を参照のこと。

- 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




```
<!-- トランザクション定義情報 -->
<tx:advice id="transactionInterceptor" >
  <tx:attributes>
    <tx:method name="updateDAO*"
               propagation="REQUIRED"
               rollback-for="java.lang.Exception"/>
    <tx:method name="*"
               propagation="REQUIRED"
               rollback-for="java.lang.RunTimeException"
               no-rollback-for="java.lang.ClassCastExeception"/>
  </tx:attributes>
</tx:advice>
```

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）

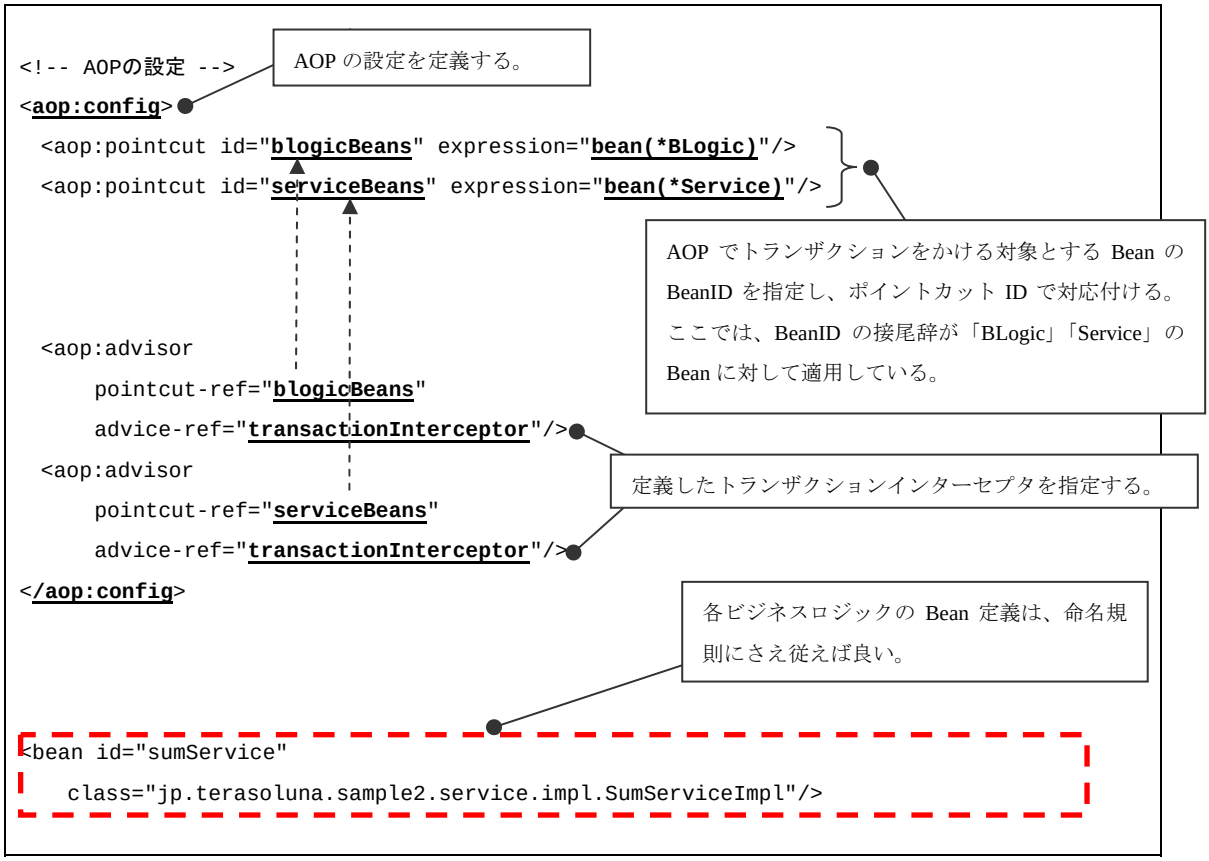
```
<!-- DataSourceの設定。 -->
<bean id="dataSource" class="……">……</bean>



<!-- 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>

<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>
```

トランザクションインターセプタを定義する。

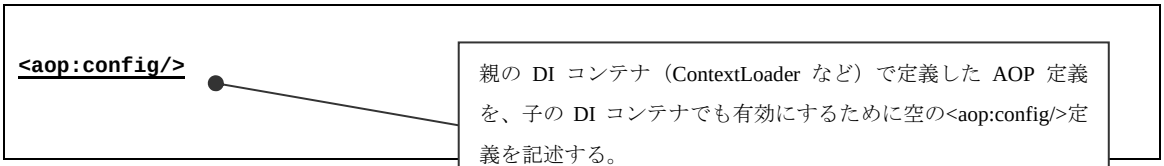
トランザクションマネージャを指定する。
省略時“transactionManager”

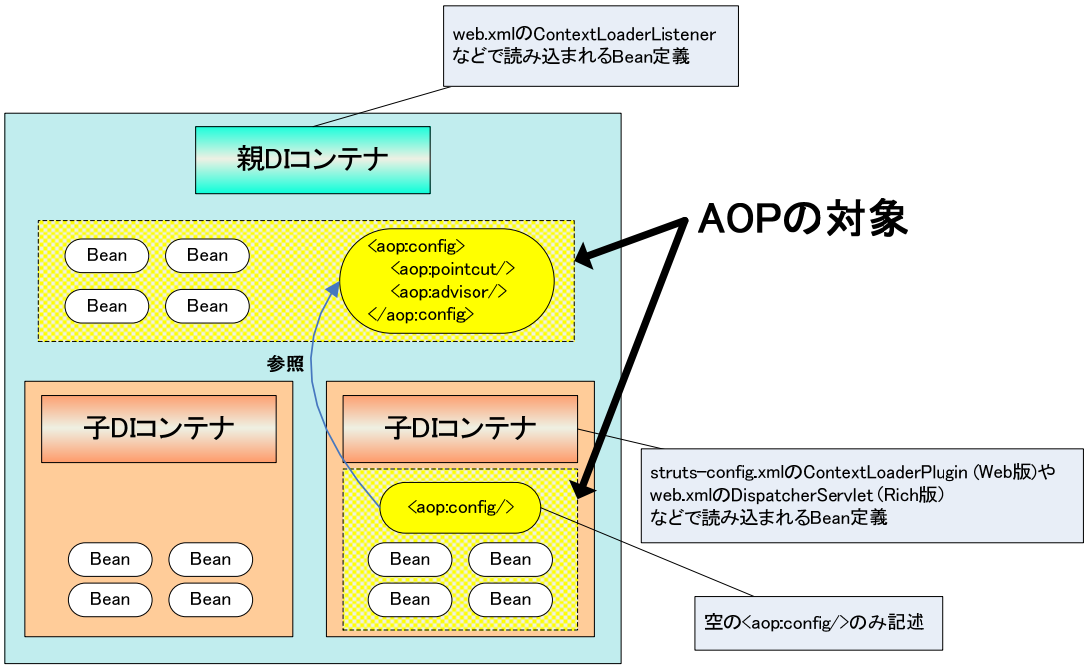
トランザクション定義情報の設定。

（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>

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

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

DAO が使用するデータソースを指定する。

- （参考）<tx:method/>の属性一覧

属性	必須	デフォルト	説明
name	○	-	トランザクション対象とするメソッド名 例：'get*' ，'handle*' ，'on*Event'
propagation	-	REQUIRED	トランザクション伝播の設定
isolation	-	DEFAULT	トランザクションの分離レベル
timeout	-	-1	トランザクションタイムアウト(秒) 無指定の場合はデフォルトタイムアウト時間が採用される。
read-only	-	false	読み込み専用トランザクションかどうか
rollback-for	-	-	ロールバック対象の例外。 カンマ区切りで複数記述できる。 デフォルトでは java.lang.RunTimeException 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();
   }
}
```

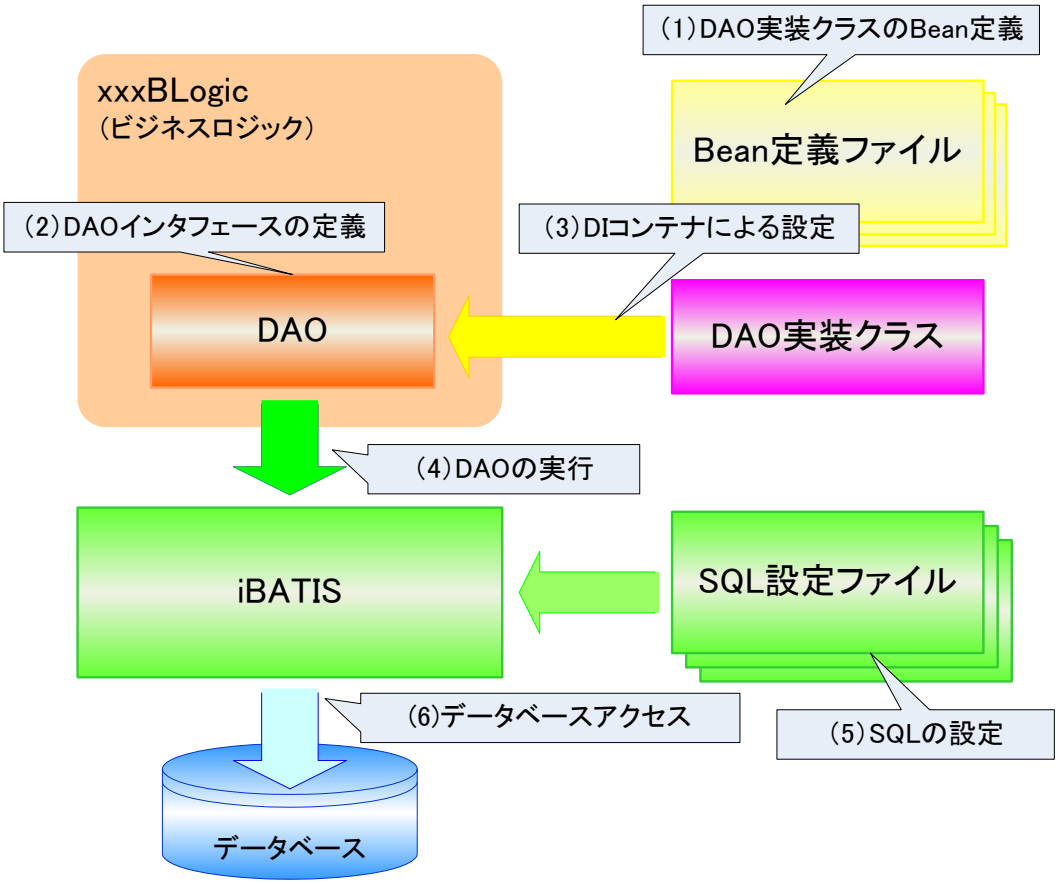
CB-01 データベースアクセス機能

■ 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">

<!-- JNDI関連のプロパティ -->
<context:property-placeholder location="WEB-INF/jndi.properties"/>
<bean id="TerasolunaDataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>${jndi.name}</value>
  </property>
</bean>
```

Property-placeholder を定義する。

context スキーマを定義する。

プロパティファイルの例 (jndi.properties)

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

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

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

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

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

Property-placeholder を定義する。

スキーマを定義する。

クラスパスを指定する。

DriverManagerDataSource の定義。

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

➢ プロパティファイルの例（jdbc.properties）

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

- iBATIS マッピング定義ファイル
  このファイルは、ビジネスロジックで利用する SQL 文と、その SQL の実行結果を JavaBean にマッピングするための定義を設定する。また、モジュール単位にファイルを作成すること。ただし、SQL の ID は、アプリケーションで一意にする必要がある。SQL の詳細な記述方法に関しては、iBATIS のリファレンスを参照のこと。
  - Select 文の実行例
    - Select 文の実行には、<select>要素を使用する。
    - resultClass 属性に SQL の結果を格納するクラスを指定する。結果が複数の場合は、指定したクラスの Collection あるいは配列が返却される。

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

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

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

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

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

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

  - <sqlMap>要素は複数記述することができるため、iBATIS マッピング定義ファイルを分割した際は、複数記述すること。

  - <settings>要素の useStatementNamespaces 属性は、SQLID を指定する際に完全修飾名で指定するかどうかを指定する。”true”を指定した場合は、『名前空間 + ”.” + SQLID』の形で SQLID を指定する。（例：名前空間が”user”、SQLID が”getUser”の場合、”user. getUser”と指定する）

- SqlMapClientFactoryBean の Bean 定義
  Spring で iBATIS を使用する場合、SqlMapClientFactoryBean を使用して iBATIS 設定ファイルの Bean 定義を DAO に設定する必要がある。SqlMapClientFactoryBean は、iBATIS のデータアクセス時に利用されるメインのクラス「SqlMapClient」を管理する役割を持つ。
  iBATIS 設定ファイルの Bean 定義はアプリケーション内で一つとする。
  - iBATIS 設定ファイル
    - “configLocation”プロパティに、iBATIS 設定ファイルのコンテキストルートからのパスを指定する。
    - 単一データベースの場合は、“dataSource”プロパティに、使用するデータソースの Bean 定義を設定する。

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

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

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

- DAO の Bean 定義
  - DAO 実装クラスは、基本的にアプリケーション Bean 定義ファイルに定義する。また、DAO もトランザクション設定対象の Bean である。トランザクションの設定方法は『CA-01 トランザクション管理機能』の機能説明書を参照のこと。
    また、Bean 定義時に DAO 実装クラスの“sqlMapClient”プロパティに iBATIS 設定ファイルの Bean 定義を設定する必要がある。

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

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

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

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

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

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

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

- QueryDAOiBatisImpl を使用した一覧データ取得例
  QueryDAOiBatisImpl を使用して、常に一覧情報（１ページ分）をデータベースから取得する場合の設定およびコーディング例を以下に記述する。

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

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

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

- Bean 定義ファイル

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

- ビジネスロジック

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

  一覧表示の詳細な使用方法は、『WI-01 一覧表示機能』を参照のこと。

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

設定された DAO を使用して、データベースから一覧情報を取得する。

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

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

設定された DAO を使用して、データベースから一覧情報を取得する。

- UpdateDAOiBatisImpl を使用したデータ登録例
  UpdateDAOiBatisImpl を使用して、データベースに情報を登録する場合の設定およびコーディング例を以下に記述する。Bean 定義ファイルの定義・設定方法は、QueryDAOiBatisImpl と同様なため省略する。
  - ビジネスロジック
    Bean 定義ファイルにて設定された DAO のメソッドを使用する。

```
private UpdateDAO updateDAO = null;
    ......Setterは省略
public void register(UserBean bean) {
    ......
    updateDAO.execute("insertUser", bean);
    ......
}
```

設定された DAO を使用して、データベースにデータを登録する。

- UpdateDAOiBatisImpl を使用した複数データの登録例（オンラインバッチ処理）
  UpdateDAOiBatisImpl を使用して、データベースに複数の情報を登録する場合の設定およびコーディング例を以下に記述する。Bean 定義ファイルの定義・設定方法は、QueryDAOiBatisImpl と同様なため省略する。
  詳細は UpdateDAOiBatisImpl の JavaDoc を参照のこと。
  - ビジネスロジック
    Bean 定義ファイルにて設定された DAO の executeBatch(List<SqlHolder>)メソッドを使用する。

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

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

  - 注意点
    executeBatch は iBATIS のバッチ実行機能を使用している。 executeBatch は戻り値として、SQL の実行によって変更された行数を返却するが、java.sql.PreparedStatement を使用しているため、 ドライバにより正確な行数が取得できないケースがある。変更行数が正確に取得できないドライバを使用する場合、 変更行数がトランザクションに影響を与える業務では (変更行数が 0 件の場合エラー処理をするケース等)、 バッチ更新は使用しないこと。
    参考資料）
    http://otndnld.oracle.co.jp/document/products/oracle11g/111/doc_dvd/java.111/E05720-02/oraperf.htm
    「標準バッチ処理の Oracle 実装の更新件数」を参照のこと。

- StoredProcedureDAOiBatisImpl を使用したデータ取得例
  StoredProcedureDAOiBatisImpl を使用して、データベースから情報を取得する場合の設定およびコーディング例を以下に記述する。Bean 定義ファイルの定義・設定方法は、QueryDAOiBatisImpl と同様なため省略する。
  - ビジネスロジック
    Bean 定義ファイルにて設定された DAO のメソッドを使用する。

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

設定された DAO を使用してプロシージャーを実行する。

  - iBATIS マッピング定義ファイル
    - プロシージャーの入出力を格納するための設定を<parameterMap>要素にて記述する。jdbcType 属性を指定すること。詳細な設定方法は、iBATIS のリファレンスを参照のこと。
      参考資料）
      http://ibatis.apache.org/docs/java/pdf/iBATIS-SqlMaps-2_ja.pdf

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

  - 実行するプロシージャー（Oracle を利用した例）

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

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

  - DataRowHandler の実装

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

public class SampleRowHandler implements DataRowHandler {
    public void handleRow(Object param) {
        if (param instanceof HogeData) {
            HogeData hogeData = (HogeData)param;
            // 一件のデータを処理するコードを記述
        }
    }
}
```

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

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

  - ビジネスロジック

```
    private QueryRowHandleDAO queryRowHandleDAO = null;
        ......Setterは省略

    public BLogicResult execute(BlogicParam params) {
        Parameter param = new Parameter();
        HogeDataRowHandler dataRowHandler = new HogeDataRowHandler();

        queryRowHandleDAO.executeWithRowHandler(
                "selectDataSql", param, dataRowHandler);

        BLogicResult result = new BLogicResult();
        result.setResultString("success");
        return result;
    }
```

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

◆ 拡張ポイント

なし。

■ 構成クラス

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

■ 関連機能

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

■ 使用例

- TERASOLUNA Server Framework for Java (Web 版) チュートリアル
  - 「2.6 データベースアクセス」
  - 「2.7 登録」
  - 一覧表示画面、登録画面

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

- TERASOLUNA Server Framework for Java (Web 版) 機能網羅サンプル
  - 「UC01 データベースアクセス」
    - /webapps/database/*
    - /webapps/WEB-INF/database/*
    - jp.terasoluna.thin.functionsample.database.*

■ 備考

- **大量データを検索する際の注意事項**

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

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

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

CC-01 JNDIアクセス機能

■ 概要

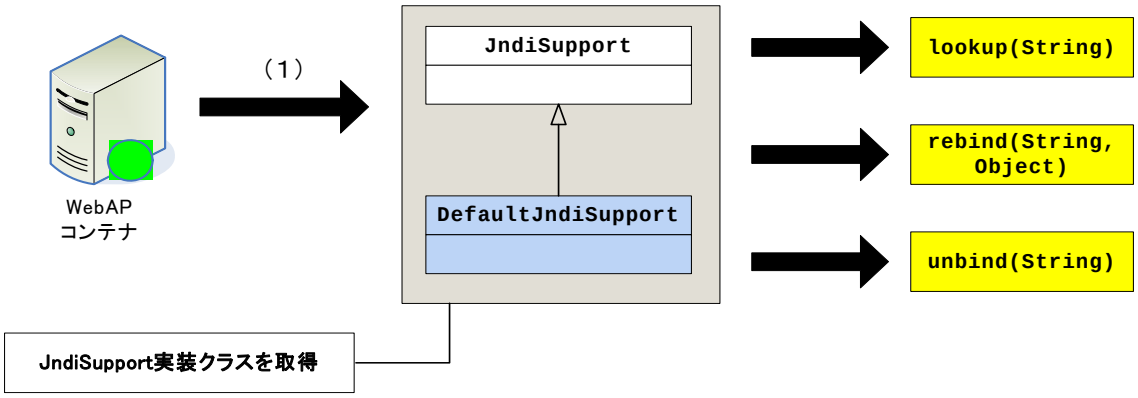
◆ 機能概要

- 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

```
<bean id="jndiSupport" scope="prototype"
  class="jp.terasoluna.fw.web.jndi.DefaultJndiSupport"
  init-method="initialize">
  <!-- セッタインジェクションで認証情報設定-->
  <property name="jndiEnvironmentMap">
    <map>
      <entry key="factory">
        <value>weblogic.jndi.WLInitialContextFactory</value>
      </entry>
      <entry key="url">
        <value>t3://localhost:7001</value>
      </entry>
      <entry key="username">
        <value>weblogic</value>
      </entry>
      <entry key="password">
        <value>password</value>
      </entry>
    </map>
  </property><!-- プロパティjndiPrefixの設定 -->
  <property name="jndiPrefix"><value>false</value></property>
</bean>
```

TERASOLUNA が提供する JndiSupport デフォルト実装クラス。

WebLogic の場合、JNDI 認証情報を設定する初期化メソッドを設定する。

JNDI ファクトリクラス名

JNDI プロバイダ URL

JNDI ユーザ名

JNDI パスワード

WebLogic のように JNDI リソース名にプリフィックス「java:comp/env/」を付けてはいけない場合、プロパティ「jndiPrefix」を false に設定する。

➢ 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 のリソースをアクセスするようになる。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。

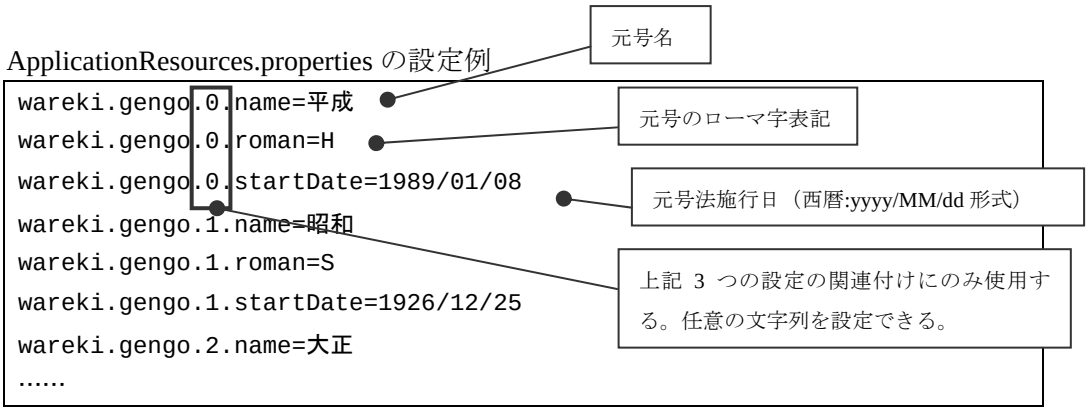
◆ ユーティリティメソッド一覧

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



- 使用例

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

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

「G」:元号を表すパターン文字。
例：
（4 個以上の連続したパターン文字）
明治、大正、昭和、平成
（3 個以下の連続したパターン文字）
M、T、S、H

「y」:年（和暦）を表すパターン文字。

「E」:曜日を表すパターン文字。
例：
（4 個以上の連続したパターン文字）
月曜日、火曜日、水曜日
（3 個以下の連続したパターン文字）
月、火、水

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

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

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

■ FileUtil

◆ 概要

ファイル操作関連のユーティリティクラス。このユーティリティクラスにて、「セッション ID に対応するディレクトリ」とは、セッション ID の 16 進ハッシュ値をディレクトリ名とするディレクトリのことを意味する。
※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
getSessionDirectoryName(String sessionId)	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。
getSessionDirectory(String sessionId)	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 temp ディレクトリを用いる。
makeSessionDirectory(String sessionId)	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、true を返す。
removeSessionDirectory(String sessionId)	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、true を返す。
rmdirs(File dir)	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.rmdirs(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=xxxxx.properties
add.property.file.3=yyyyy.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 null のときは null 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 null のときは null 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 null のときは null 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 delim)	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。 csvString が null 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。csvString が null 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。
dump(Map<?, ?> map)	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 toString()を行い、文字列を出力する。
getArraysStr(Object[] arrayObj)	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。
hankakuToZenkaku(String value)	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。
zenkakuToHankaku(String value)	全角文字列を半角文字列に変換する。
filter(String str)	HTML メタ文字列に変換する。
toLikeCondition(String condition)	検索条件文字列を LIKE 述語のパターン文字列に変換する。
getByteLength(String value, String encoding)	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", extenthin2 = ".xls"）
String extention1 = getExtention("Test.txt");
String extention2 = getExtention("Test1.Test2.xls");

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

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

（続く）

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

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

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

// 全角文字列を半角文字列に変換する。（結果：hankaku = "A!ｱ"）
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.getByteLength("あああ", "UTF-8");
```

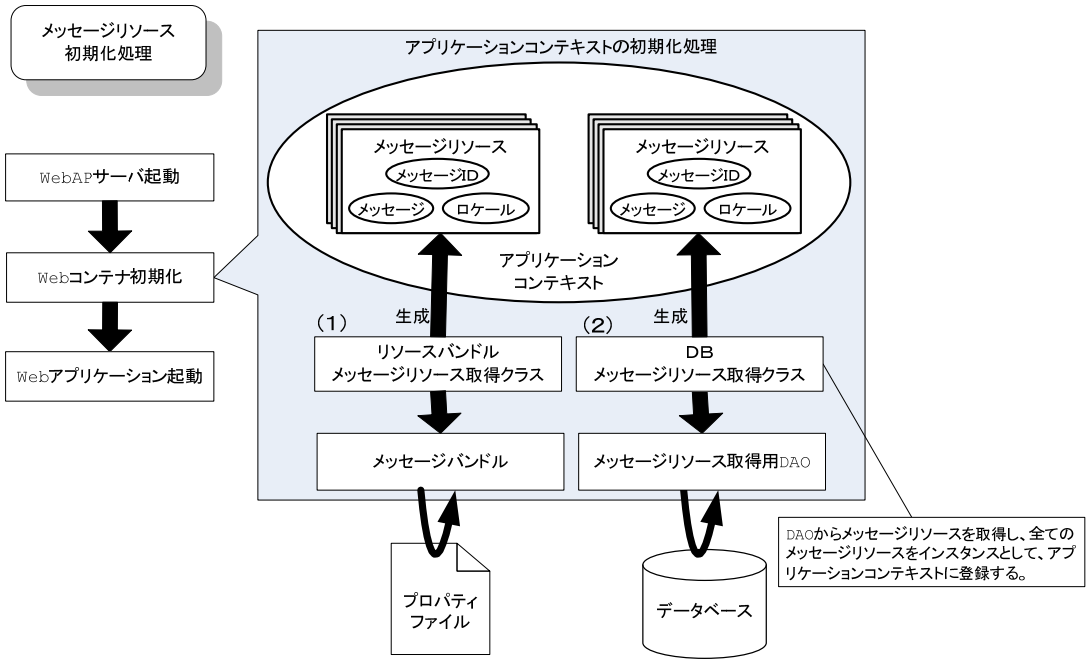
CE-01 メッセージ管理機能

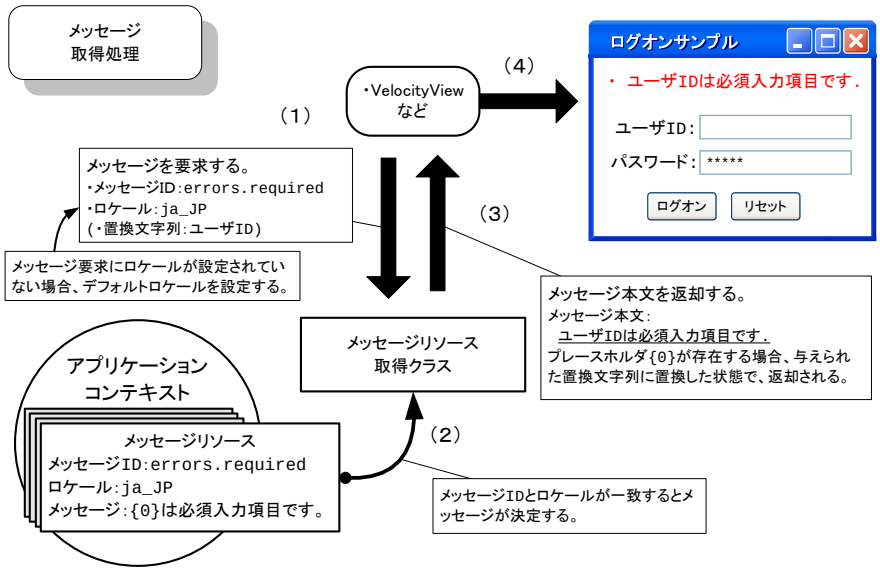
■ 概要

◆ 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処理

(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 DBMessageRecourceDAOImpl 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 するためのセッターを記述する。

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。

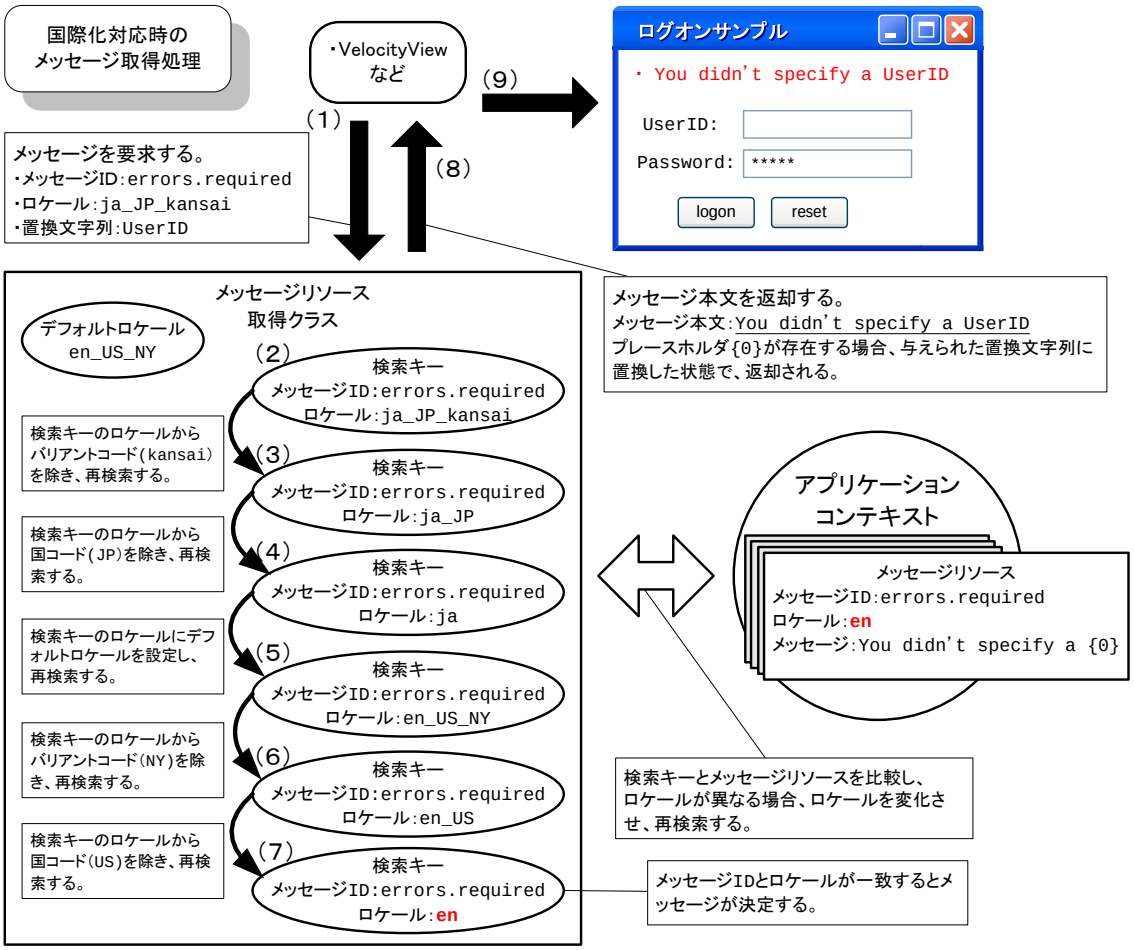
jp.terasoluna.fw.message.DataSourceMessageSource クラスの reloadDataSourceMessage メソッド

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アントコードがある場合、バリアントコードを除き、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アントコードがある場合、

バリアントコードを除き、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
・バリアントコードカラム

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、バリアント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アント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	バリア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), (バリアントコード), メッセージ本体 FROM テーブル名 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</value>
  <property name=variantColumn value="HOUGEN</value>
  <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。
バリアントコードのカラム名を指定す

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
メッセージのバリア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
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

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

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 bean="dataSource"/>
  <property name="tableName" value="DBMESSAGES"/>
  <property name="codeColumn" value="BANGOU"/>
  <property name="messageColumn" 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 bean="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。

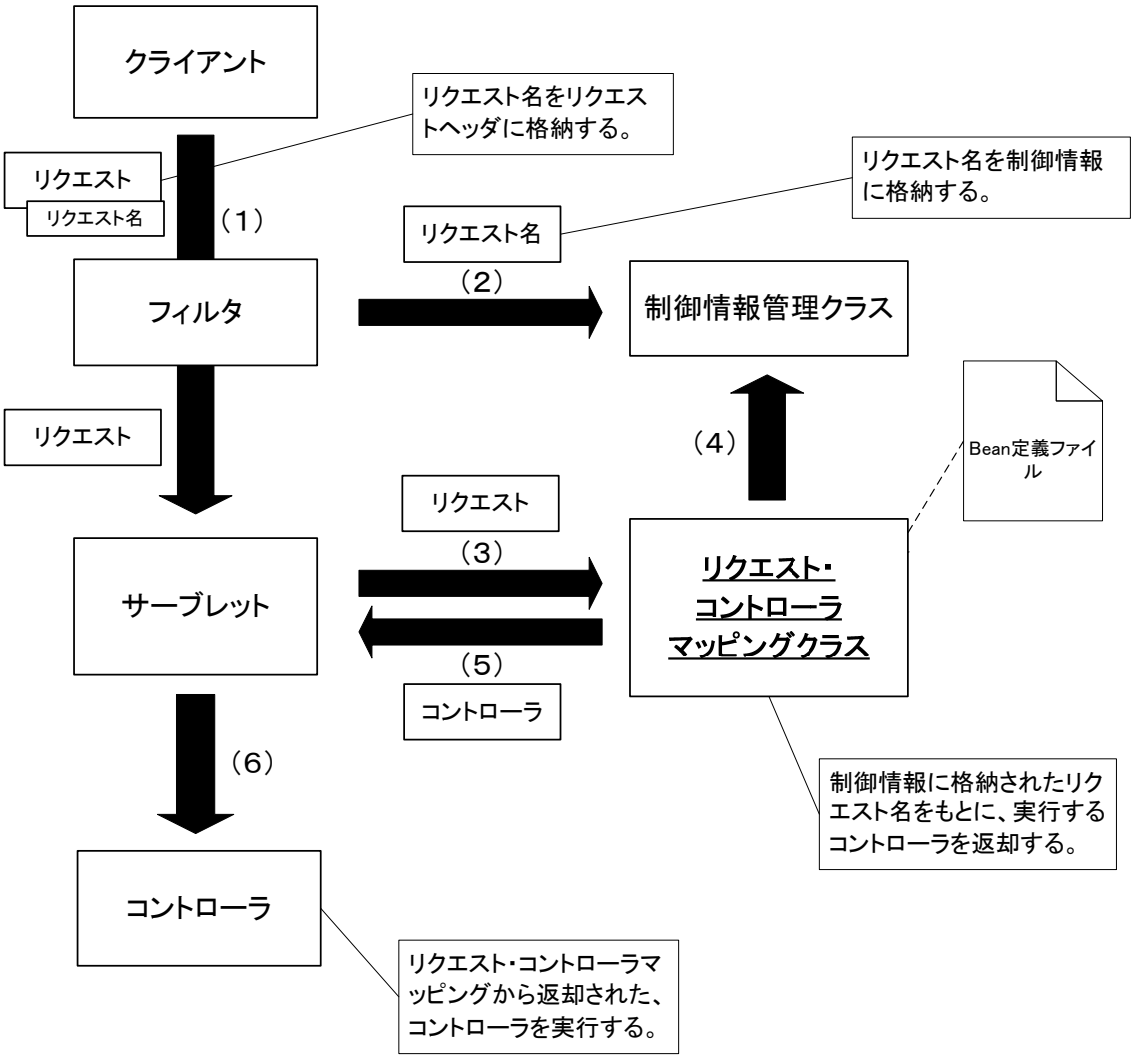
RA-01 リクエスト・コントローラマッピング機能

■ 概要

◆ 機能概要

- リクエストと実行するコントローラとの対応付けを行う機能である。
  - リクエストヘッダのリクエスト名に基づいて、実行するコントローラが決定される。
  - リクエストのヘッダ以外からリクエスト名を取得する場合、TERASOLUNA Server Framework for Java (Rich 版)の拡張を行なう必要がある。

◆ 概念図



◆ 解説

(1) ブラウザからリクエストをサーバに送信する。
(2) フィルタがリクエストヘッダからリクエスト名を取得し、制御情報に格納する。
(3) サーブレット（ディスパッチャサーブレット）がリクエスト・コントローラマッピングを実行する。
(4) リクエスト・コントローラマッピングクラスは、制御情報管理クラスを使用して、制御情報に格納されたリクエスト名を取得する。
(5) (4)で取得したリクエスト名とBean定義ファイルに設定された接頭辞・接尾辞をもとに、コントローラを取得し、サーブレットに返却する。
(6) サーブレットは返却されたコントローラを実行する。

■ 使用方法

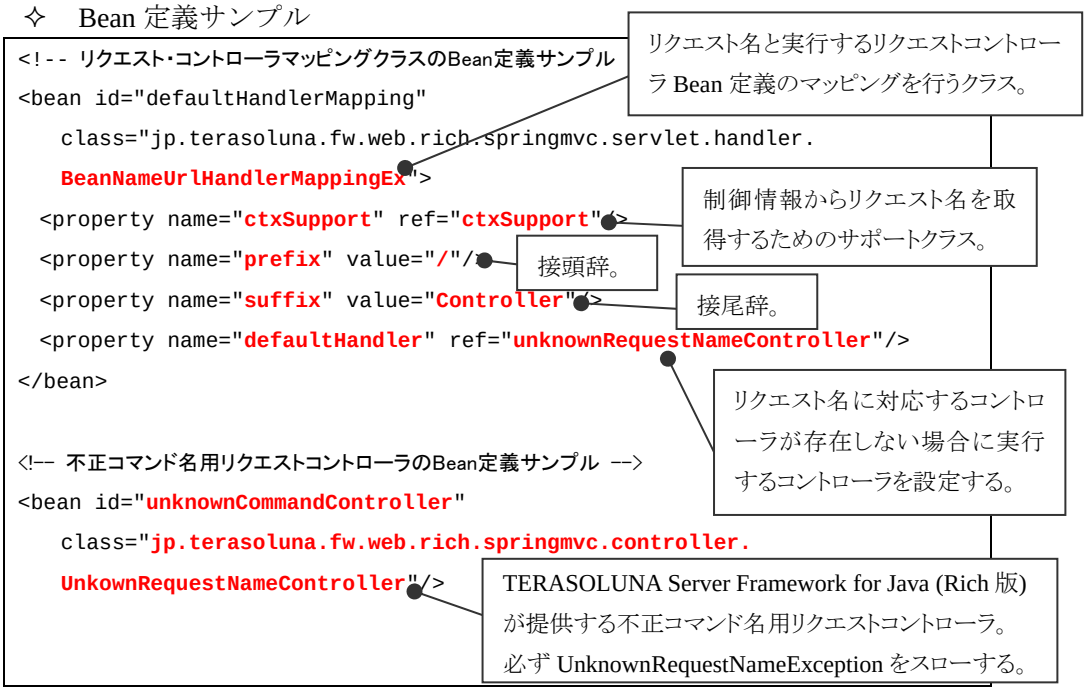
◆ コーディングポイント

- 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
  - ➢ リクエスト・コントローラマッピングクラスの Bean 定義
    以下のプロパティを設定する。

	プロパティ名	必須	概要
1	ctxSupport	○	リクエスト名を取得するためのサポートクラス。 詳細は、『RB-04 制御情報管理機能』参照のこと。
2	prefix	×	コントローラ ID の接頭辞。
3	suffix	×	コントローラ ID の接尾辞
4	defaultHandler	○	リクエスト名に対応するコントローラが存在しない場合に実行するコントローラ。

  - ✧ コントローラ ID の生成ルール
    コントローラ ID は『接頭辞＋リクエスト名＋接尾辞』の式で求められる。
    （例）リクエスト名が“sum”、接頭辞が“/”、接尾辞が“Controller”の場合、コントローラ ID は「/sumController」となる。

✧ Bean 定義サンプル

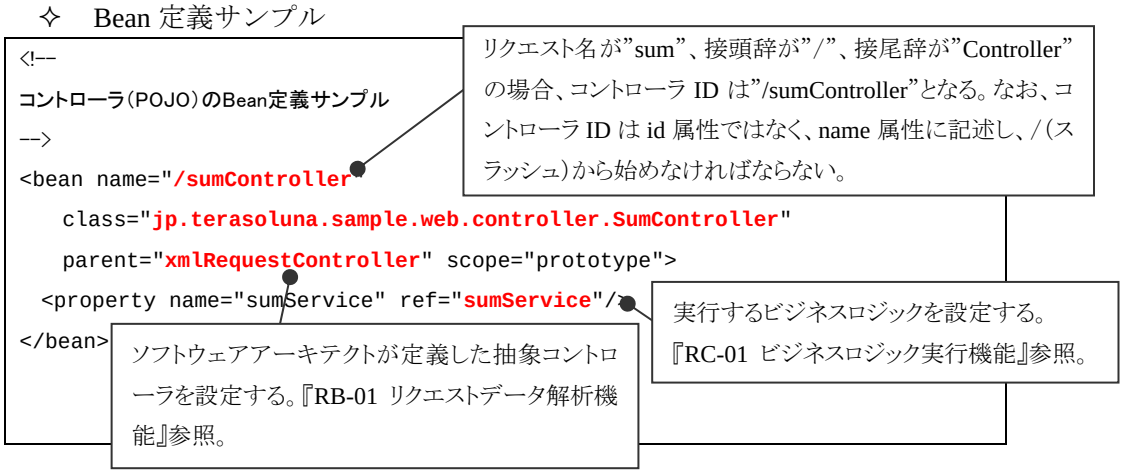



```
<!-- リクエスト・コントローラマッピングクラスのBean定義サンプル
<bean id="defaultHandlerMapping"
    class="jp.terasoluna.fw.web.rich.springmvc.servlet.handler.
    BeanNameUrlHandlerMappingEx">
  <property name="ctxSupport" ref="ctxSupport"/>
  <property name="prefix" value="/"/>
  <property name="suffix" value="Controller"/>
  <property name="defaultHandler" ref="unknownRequestNameController"/>
</bean>

<!-- 不正コマンド名用リクエストコントローラのBean定義サンプル -->
<bean id="unknownCommandController"
    class="jp.terasoluna.fw.web.rich.springmvc.controller.
    UnkownRequestNameController"/>
```

➢ フィルタ（RequestContextHandlingFilter）の定義。
リクエストヘッダからリクエスト名を取得するフィルタを、デプロイメントディスクリプタに定義する。

✧ 詳細は、『RB-04 制御情報管理機能』を参照のこと。

- 業務開発者が行うコーディングポイント
  - コントローラの Bean 定義
    リクエスト名に対応するコントローラの Bean 定義を行う。
    - Bean 定義サンプル




```
<!--
コントローラ（POJO）のBean定義サンプル
-->
<bean name="/sumController"
    class="jp.terasoluna.sample.web.controller.SumController"
    parent="xmlRequestController" scope="prototype">
  <property name="sumService" ref="sumService"/>
</bean>
```

■ リファレンス

◆ 構成クラス

	クラス名	概要
1	BeanNameUrlHandlerMappingEx	リクエスト名と実行するリクエストコントローラのマッピングを行うクラス。
2	RequestContexthandlingFilter	制御情報の生成・破棄を行うフィルタ。
3	UnkownRequestNameController	TERASOLUNA Server Framework for Java (Rich 版) が提供するコントローラ。 必ず UnknownRequestNameException をスローする。
4	TerasolunaController	サービス層のクラスを実行するリクエストコントローラ抽象クラス。

◆ 拡張ポイント

なし

■ 関連機能

- 『RA-02 コントローラ拡張機能』
- 『RB-01 リクエストデータ解析機能』
- 『RB-04 制御情報管理機能』
- 『RC-01 サービスクラス実行機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Rich 版) **機能網羅サンプル**
  - UC101 **リクエスト・コントローラマッピング**
    - jp.terasoluna.rich.functionsample.controllermapping.*

- TERASOLUNA Server Framework for Java (Rich 版) チュートリアル
  - 「2.3 単純なロジック」
  - /webapps/WEB-INF/applicationContext.xml
  - /webapps/WEB-INF/tutorial-controller.xml 等

■ 備考

- なし

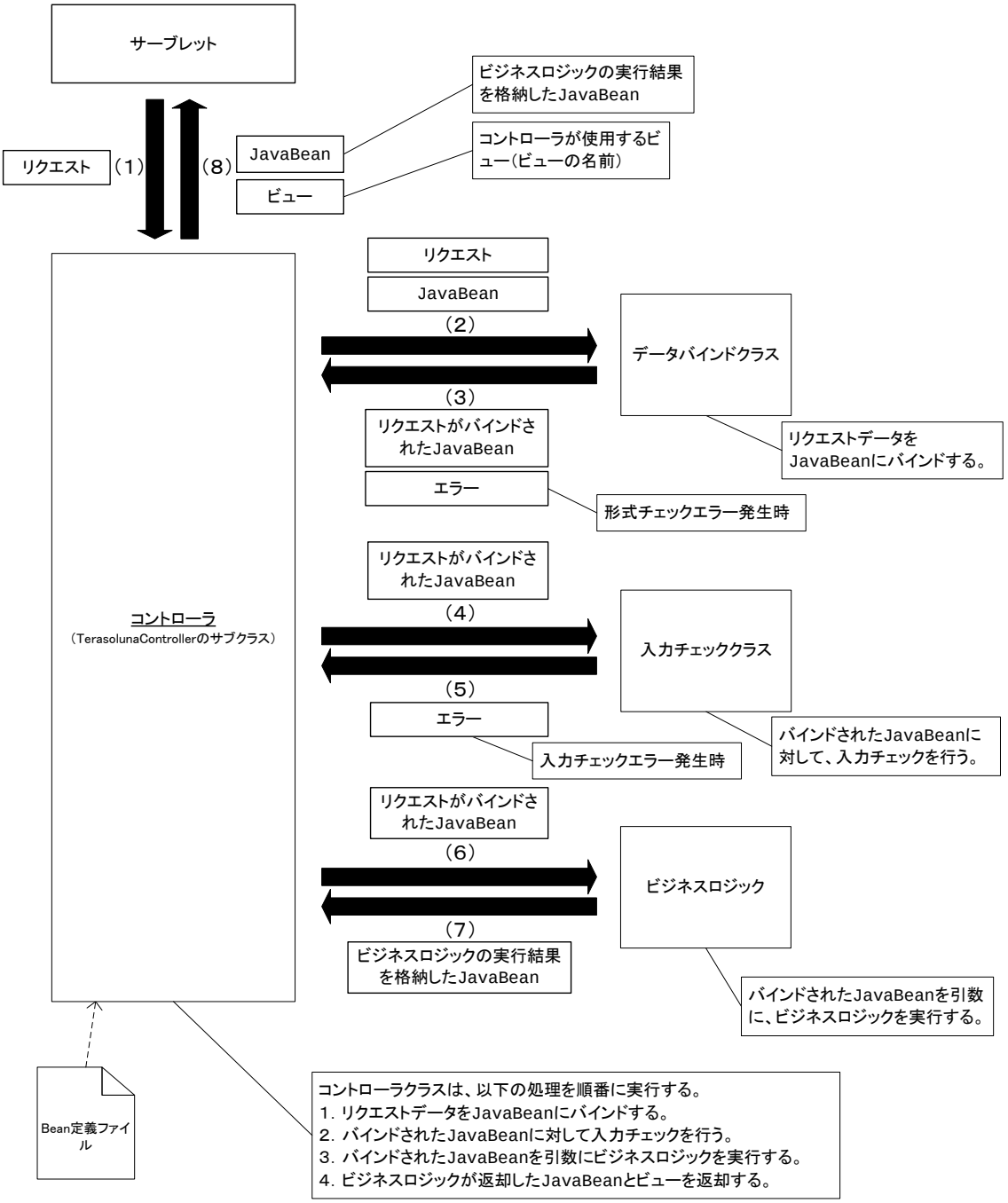
RA-02 コントローラ拡張機能

■ 概要

◆ 機能概要

- サービス層のクラスを実行するリクエストコントローラとして、以下の機能を持った抽象クラスを定義する。
  - ① HTTPリクエストをJavaBeanにバインドする機能
    - ✧ XML形式とクエリ形式のいずれかのリクエストデータをJavaBeanにバインドする。
  - ② 入力チェック機能
    - ✧ リクエストデータがバインドされた JavaBean に対して入力チェックを行う。
  - ③ サービス実行機能
    - ✧ リクエストデータがバインドされた JavaBean を引数として、ビジネスロジックを実行する。
  - ④ JavaBeanとビューを返却する機能
    - ✧ ビジネスロジックから返却されたJavaBeanとビューをサーブレットに返却する。

◆ 概念図



◆ 解説

(1) サーブレット（ディスパッチャサーブレット）は、リクエストを引数にコントロー

ラ（TerasolunaControllerのサブクラス）を実行する。
(2) コントローラはリクエストデータとJavaBeanを用い、データバインドを実行する。
(3) データバインドクラスは、リクエストデータをJavaBeanにバインドして返却する。
- XML 形式のリクエストデータを形式チェックする Bean 定義が行われていた場合、データバインドクラスは形式チェックを行う。
  - 形式チェックエラーが発生した場合、エラー情報を生成し、コントローラに返却する。
  - 詳細は、『RB-01 リクエストデータ解析機能』、『RF-01 形式チェック機能』を参照のこと。
- エラー情報が返却された場合、コントローラは例外処理を行なう。
  - 詳細は、『RD-01 例外処理機能』を参照のこと。

(4) コントローラは、リクエストデータがバインドされたJavaBeanを引数に、入力チェックを実行する。
(5) 入力チェッククラスは、引数で渡されたJavaBeanに対して、入力チェックを行う。
- 入力チェックエラーが発生した場合、エラー情報を生成し、コントローラに返却する。
  - 詳細は、『RF-02 入力チェック機能』を参照のこと。
- エラー情報が返却された場合、コントローラは例外処理を行なう。
  - 詳細は、『RD-01 例外ハンドリング機能』を参照のこと。

(6) コントローラは、リクエストデータがバインドされたJavaBeanを引数に、ビジネスロジックを実行する。
(7) ビジネスロジックは、実行結果をJavaBeanとして返却する。
- 詳細は、『RC-01 ビジネスロジック実行機能』を参照のこと。

(8) コントローラは、(7)で返却されたJavaBeanと、コントローラに対応するビューを返却する。
- 返却されたビューを使用して、サーブレット（ディスパッチャサーブレット）はレスポンスデータを生成する。
  - 詳細は、『RB-02 レスポンスデータ生成機能』を参照のこと。

■ 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
  - 制御情報管理クラスの Bean 定義
    - ✧ 詳細は、『RB-04 制御情報管理機能』を参照すること。
  - リクエストデータバインドクラスの Bean 定義
    - ✧ 詳細は、『RB-01 リクエストデータ解析機能』を参照すること。
  - 入力チェッククラスの Bean 定義
    - ✧ 詳細は、『RF-02 入力チェック機能』を参照すること。
  - レスポンスデータ生成クラスの Bean 定義
    - ✧ 詳細は、『RB-02 レスポンスデータ生成機能』を参照すること。
  - コントローラ抽象 Bean 定義
    - ✧ リクエストを受信するコントローラの親となるコントローラの抽象 Bean 定義を有効化する。詳細はリファレンスの『コントローラ抽象 Bean 定義』を参照のこと。

- Bean 定義サンプル

```
<!-- ==================== POJOコントローラの定義 ==================== -->
<!-- コントローラ抽象Bean定義
 （ビジネスロジック：POJO）
-->
<bean id="pojoController" abstract="true">
  <property name="ctxSupport" ref="ctxSupport" />
  <property name="validator" ref="be
</bean>
<!-- コントローラの抽象Bean定義
 （ビジネスロジック：POJO、受信リクエスト：XML形式）
-->
<bean id="pojoXmlRequestController" abstract="true"
    parent="pojoController">
  <property name="dataBinderCreator" ref
</bean>
<!-- コントローラの抽象Bean定義
 （ビジネスロジック：POJO、受信リクエスト：XML形式、ビュー：Castor）
-->
<bean id="pojoXmlRequestCastorViewController" abstract="true"
   parent="pojoXmlRequestController"/>
<!-- コントローラの抽象Bean定義
 （ビジネスロジック：POJO、受信リクエスト：XML形式、ビュー：Velocity）
-->
<!--
<bean id="pojoXmlRequestVelocityViewController" abstract="true"
    parent="pojoXmlRequestController">
  <property name="useRequestNameView" value="true"/>
</bean>
-->
    ......
```

親として定義されている、Bean 定義を有効にする。

親として定義されている、Bean 定義を有効にする。

利用する Bean 定義を有効にする。

利用しない Bean 定義は無効にする。

- 業務開発者が行うコーディングポイント
  - リクエストに対応するコントローラの Bean 定義
    - 詳細は、『RC-01 ビジネスロジック実行機能』を参照すること。
  - ビジネスロジックの Bean 定義
    - 詳細は、『RC-01 ビジネスロジック実行機能』を参照すること。

■ リファレンス

◆ 構成クラス

	クラス名	概要
1	TerasolunaController	サービス層のクラスを実行するリクエストコントローラ抽象クラス。
2	BLogicController	BLogic インタフェース実装クラス実行用リクエストコントローラ。
3	DispatcherServlet	一連の処理フローを実行するサーブレット。リクエストとコントローラのマッピング、コントローラの実行、ビューの生成を行う。
4	ServletRequestDataBinderCreator	ServletRequestDataBinder を生成するクラスが実装すべきインタフェース。
5	QueryServletRequestDataBinderCreator	クエリ形式で定義されたリクエストデータをバインドするクラス（ServletRequestDataBinder）を生成する ServletRequestDataBinderCreator 実装クラス。
6	XMLServletRequestDataBinderCreator	XML 形式で定義されたリクエストデータをバインドするクラス（XMLServletRequestDataBinder）を生成する ServletRequestDataBinderCreator 実装クラス。
7	ServletRequestDataBinder	クエリ形式のリクエストデータを業務入力 JavaBean にバインドする DataBinder 継承クラス。
8	XMLServletRequestDataBinder	XML 形式のリクエストデータを業務入力 JavaBean にバインドする ServletRequestDataBinder 継承クラス。
9	BLogic	BLogicController を使用するときに、サービス層のクラスが実装すべきインタフェース。
10	ModelAndView	ビジネスロジックの実行結果として返却される JavaBean とビューを保持するクラス。
11	Validator	入力チェックを行うクラスが実装すべきインタフェース

◆ コントローラ抽象Bean定義

➢ デフォルトで用意している、コントローラの抽象 Bean 定義の一覧を示す。“1”, “2”, “6”, “10”, “11”, “15”は親となる抽象 Bean 定義である。例えば、“7”の pojoQueryRequestCastorViewController を使用する場合は、“1”の pojoController と“6”の pojoQueryRequestController を有効化する必要がある。

	Bean id	ビジネスロジック	受信リクエスト	ビュー
1	pojoController	POJO	-	-
2	pojoXmlRequestController	POJO	XML 形式	-
3	pojoXmlRequestCastorViewController	POJO	XML 形式	Castor
4	pojoXmlRequestVelocityViewController	POJO	XML 形式	Velocity
5	pojoXmlRequestDefaultFileDownloadViewController	POJO	XML 形式	DefaultFileDownloadView
6	pojoQueryRequestController	POJO	クエリ形式	-
7	pojoQueryRequestCastorViewController	POJO	クエリ形式	Castor
8	pojoQueryRequestVelocityViewController	POJO	クエリ形式	Velocity
9	pojoQueryRequestDefaultFileDownloadViewController	POJO	クエリ形式	DefaultFileDownloadView
10	blogicController	BLogic	-	-
11	blogicXmlRequestController	BLogic	XML 形式	-
12	blogicXmlRequestCastorViewController	BLogic	XML 形式	Castor
13	blogicXmlRequestVelocityViewController	BLogic	XML 形式	Velocity
14	blogicXmlRequestDefaultFileDownloadViewController	BLogic	XML 形式	DefaultFileDownloadView
15	blogicQueryRequestController	BLogic	クエリ形式	-
16	blogicQueryRequestCastorViewController	BLogic	クエリ形式	Castor
17	blogicQueryRequestVelocityViewController	BLogic	クエリ形式	Velocity
18	blogicQueryRequestDefaultFileDownloadViewController	BLogic	クエリ形式	DefaultFileDownloadView

◆ 拡張ポイント

- TerasolunaController は、サービスクラス実行処理の前処理・後処理を実装するための拡張ポイントを用意している。
  - ✧ 前処理・後処理を実装することで、セッションから JavaBean への反映等の処理を追加することが可能である。
  - ✧ 前処理を実装する場合、preService()メソッドを、後処理を実装する場合、postService()メソッドをオーバライドする。

■ 関連機能

- 『RA-01 リクエスト・コントローラマッピング機能』
- 『RB-01 リクエストデータ解析機能』
- 『RB-02 レスポンスデータ生成機能』
- 『RB-04 制御情報管理機能』
- 『RC-01 ビジネスロジック実行機能』

- 『RD-01 例外ハンドリング機能』
- 『RF-02 入力チェック機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Rich 版) 機能網羅サンプル
  - UC111 コントローラ拡張
    - ✧ jp.terasoluna.rich.functionsample.controllerex.*

- TERASOLUNA Server Framework for Java (Rich 版) チュートリアル
  - 「2.3 単純なロジック」
  - /webapps/WEB-INF/tutorial-servlet.xml 等

■ 備考

- なし

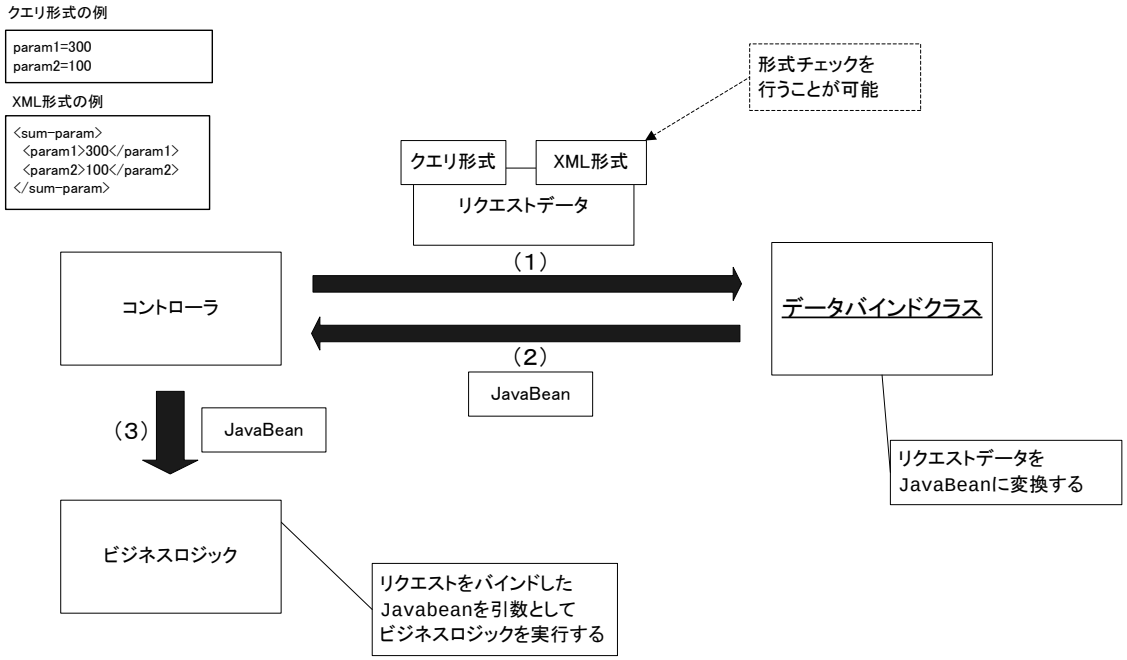
RB-01 リクエストデータ解析機能

■ 概要

◆ 機能概要

- リクエストデータを JavaBean にバインドする機能である。
  この JavaBean はビジネスロジックの引数として利用される。
- リクエストデータを JavaBean にバインド可能なリクエストデータとしては、クエリ形式と XML 形式の２種類がある。
- クエリ形式と XML 形式は、リクエストデータごとに切り替えが可能であり、Bean 定義ファイルにて定義する。
- リクエストデータが XML 形式の場合のみ、XML スキーマによる形式チェックを行うことができる。
  - 形式チェックを行うメリットとして、クライアントに XML の形式エラー情報を返却することができる。
  - 形式チェックを行うデメリットとして、パフォーマンスが劣化する。
  - 形式チェックの有無は Bean 定義ファイルに設定し、システムで一意となる。
  - 上記設定を行わない場合、デフォルトで形式チェックを行う設定となる。

◆ 概念図



◆ 解説

(1) コントローラは受信したリクエストデータとバインド先のJavaBeanを引数に、データバインドを実行する。
Bean定義に従い、クエリ形式、またはXML形式のデータバインドを切り替える。

(2) データバインドクラスは、リクエストデータの形式チェックを行い、JavaBeanに変換して返却する。
形式チェック実行時にエラーが発生した場合、コントローラにエラーを返却する。

- XML形式のリクエストデータの形式チェック
  XMLスキーマによる形式チェックを行う。詳細は、『RF-01 形式チェック機能』を参照のこと。
- クエリ形式のリクエストデータの形式チェック
  Springから標準で提供される機能を用いて形式チェックを行う。詳細は、『コーディングポイント』を参照のこと。

(3) コントローラはデータバインドで生成されたJavaBeanを引数に、業務ロジックを起動する。
2でエラーが返却された場合、コントローラは例外のハンドリングを行う。

- 詳細は、『RD-01 例外ハンドリング機能』を参照のこと。

■ 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
  Bean 定義の設定すべきポイントとして、下記の項目が挙げられる。
  1. リクエストを受信するコントローラの親となるコントローラの抽象 Bean 定義を有効化する。詳細は『RA-02 コントローラ拡張機能』を参照のこと。
  2. 受信リクエストに XML 形式を利用し、形式チェックを行う場合、XML データの形式チェックを行う SchemaValidator デフォルト実装クラスを定義する。

- Bean 定義サンプル

```
<!-- OXMapper実装クラスのBean定義サンプル-->
<bean id="oxmapper"
class="jp.terasoluna.fw.oxm.mapper.castor.CastorOXMapperImpl" />

<!-- SchemaValidatorのBean定義サンプル（※名前空間を使用しない場合） -->
<bean id="schemaValidator"
class="jp.terasoluna.fw.oxm.xsd.xerces.SchemaValidatorImpl"/>

<!-- SchemaValidatorのBean定義サンプル（※名前空間を使用する場合） -->
<!--
<bean id="schemaValidator"
class="jp.terasoluna.fw.oxm.xsd.xerces.SchemaValidatorImpl"
init-method="initNamespaceProperties">
  <property name="namespace" value="true"/>
</bean>
-->
```

スキーマ定義ファイルに名前空間を宣言する場合の設定。

- 業務開発者が行うコーディングポイント
  - ➢ コントローラの Bean 定義

    受信するリクエスト形式に応じて、ソフトウェアアーキテクトが有効にした抽象 Bean 定義を、親のコントローラに設定する。
  - ➢ Bean 定義サンプル

```
<!--
コントローラのBean定義サンプル(受信リクエスト：クエリ形式)
-->
<bean name="/sumController"
    class="jp.terasoluna.sample2.web.controller.SumController"
    parent=" pojoQueryRequestCastorViewController " scope="prototype">
    <property name="sumService" ref="sumService"/>
</bean>

<!--
コントローラのBean定義サンプル(受信リクエスト：XML形式)
-->
<bean name="/maxController"
    class="jp.terasoluna.sample2.web.controller.MaxController"
    parent=" pojoXmlRequestCastorViewController " scope="prototype">
    <property name="sumService" ref="sumService"/>
</bean>
```

ソフトウェアアーキテクトが有効にした抽象 Bean 定義を、クエリ形式のリクエストデータを受信するコントローラに設定する。

ソフトウェアアーキテクトが有効にした抽象 Bean 定義を、XML 形式のリクエストを受信するコントローラに設定する。

➢ クエリ形式の形式チェック・バインド定義

基本データ型、基本データ型のラッパークラス、String 型、String 配列型、BigDecimal 型、JavaBean 型の属性にバインドを行う場合、特別な定義を行う必要はない。ただし、日付型・Collection 型の属性にバインドを行う場合、Spring が提供するプロパティエディタを各コントローラに登録する必要がある。
リクエストデータに対する形式チェックは自動的に行われる。形式チェックで発生するエラーの詳細は、『リファレンス』を参照すること。

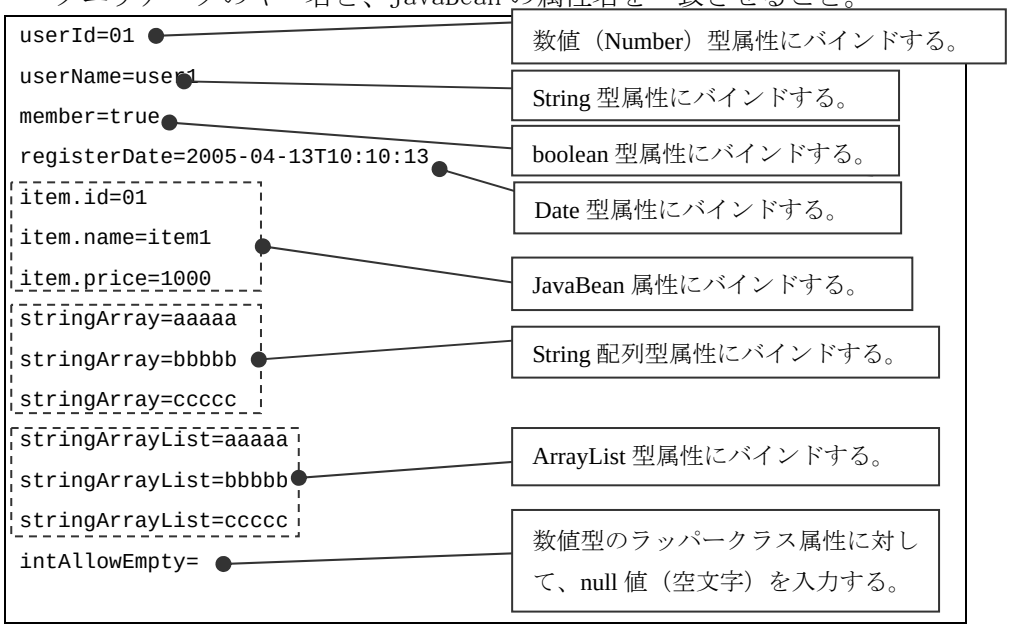
✧ プロパティエディタの機能

➢ データ型の変換
リクエストデータの文字列を、Date 型や ArrayList 型の属性にバインドするために型変換を行う。

➢ 値のフォーマット
数値型、Date 型の属性をフォーマットすることが可能。
例）new CustomDateEditor(
SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss"), false));

➢ null 値（空文字）の許容
入力値として null 値（空文字）を許容するかどうかの設定を行う。
null 値を許容した場合、初期値として null が設定される。
本機能に対応するデータ型の詳細は、『リファレンス』を参照すること。
※入力値に null 値を許容する場合、コンストラクタの引数 allowEmpty に true を設定してプロパティエディタを生成すること。
例）CustomNumberEditor(Integer.class, **true**);

✧ クエリデータサンプル
クエリデータのキー名と、JavaBean の属性名を一致させること。



✧ 形式チェック・マッピング対象の JavaBean サンプル（getter,setter は省略）

```
public class SampleDto3 {
  private int userId;
  private String userName;
  private boolean member;
  private Date registorDate = null;
  private Item2 item = new Item2();
  private List<String> stringList = null;
  private ArrayList<String>  stringArrayList = null;
 private Integer intAllowEmpty;;
}

public class Item2 {
  private int id;
  private String name;
  private int price;
}
```

✧ コントローラサンプル

initBinder メソッド内にプロパティエディタを登録する。
プロパティエディタの詳細は、『リファレンス』を参照のこと。

```
public class SumController
extends TerasolunaController<SumParam, SumResult> {
  private SumService sumService;

  @Override
  public void initBinder(HttpServletRequest req, ServletRequestDataBinder
binder) {
    //Date型の属性に対してプロパティエディタを登録する。
    SimpleDateFormat format= new SimpleDateFormat("yyyy-MM-
dd'T'HH:mm:ss");
    binder.registerCustomEditor(Date.class, "registerDate", new
CustomDateEditor(format, false));
    //ArrayList型の属性に対してプロパティエディタを登録する。
    binder.registerCustomEditor(ArrayList.class, "stringArrayList", new
CustomCollectionEditor(ArrayList.class, false));
    //Number（数値）型の属性に対してプロパティエディタを登録する。
    binder.registerCustomEditor(Integer.class, "intTest01", new
CustomNumberEditor(Integer.class, true));
  }
public void setSumService(SumService sumService) {
  this.sumService = sumService;
}

  @Override
  protected SumResult executeService(SumParam command) throws Exception {
    return sumService.sum(command);
  }
}
```

フォーマットを指定する。

Date 型のプロパティエディタ。

Collection 型のプロパティエディタ。

数値型の
プロパティエディタ。

属性に対して、null 値（空文字）の入力を許容する場合、引数 AllowEmpty に true を設定する。

➢ XML 形式の形式チェック・マッピング定義

形式チェックに関しての詳細は、『RF-01 形式チェック機能』を、
マッピングに関しての詳細は、『RB-03 XML-Object 変換機能』を参照すること。

■ リファレンス

◆ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.rich.springmvc.bind.creator.ServletRequestDataBinderCreator	ServletRequestDataBinder を生成するクラスが実装すべきインタフェース。
2	jp.terasoluna.fw.web.rich.springmvc.bind.creator.QueryServletRequestDataBinderCreator	クエリ形式で定義されたリクエストデータをバインドするクラスを生成する ServletRequestDataBinderCreator 実装クラス。
3	jp.terasoluna.fw.web.rich.springmvc.bind.creator.XMLServletRequestDataBinderCreator	XML 形式で定義されたリクエストデータをバインドするクラスを生成する ServletRequestDataBinderCreator 実装クラス。
4	jp.terasoluna.fw.web.rich.springmvc.bind.XMLServletRequestDataBinder	XML 形式のリクエストデータを業務入力 JavaBean にバインドするクラス。
5	jp.terasoluna.fw.oxm.mapper.OXMapper	Object-XML 変換を行うためのインタフェース。
6	jp.terasoluna.fw.oxm.mapper.castor.CastorOXMapperImpl	Castor の機能を利用して XML-Object 変換を行う OXMapper デフォルト実装クラス。
7	jp.terasoluna.fw.oxm.xsd.SchemaValidator	XML データの形式チェックを行うクラスが実装すべきインタフェース。
8	jp.terasoluna.fw.oxm.xsd.xerces.SchemaValidatorImpl	XML データの形式チェックを行う SchemaValidator デフォルト実装クラス。
9	java.beans.PropertyEditor	プロパティエディタのインタフェース。
10	java.beans.PropertyEditorSupport	プロパティエディタの基底クラス。
11	org.springframework.beans.propertyeditors.CustomNumberEditor	数値型のプロパティエディタ 数値型への型変換、値のフォーマット、null 値（空文字）を許容する設定を行う。
12	org.springframework.beans.propertyeditors.CustomBooleanEditor	boolean 型のプロパティエディタ boolean 型への型変換、null 値（空文字）を許容する設定を行う。
13	org.springframework.beans.propertyeditors.CustomD	Date 型のプロパティエディタ Date 型への型変換、値のフォーマット、null 値（空文字）を許容する

	ateEditor	設定を行う。
14	org.springframework.beans.propertyeditors.CustomCollectionEditor	Collection 型のプロパティエディタ Collection 型への型変換を行う、null 値(空文字)を許容する設定を行う。

◆ プロパティエディタ

➢ バインドするデータ型と、使用するプロパティエディタの一覧を記す。

	データ型	プロパティエディタ	備考
1	**基本データ型**	CustomNumberEditor	基本的にはプロパティエディタを設定する必要はない。 null 値を許容することはできない。
2	**基本データ型のラッパークラス(数値)**	CustomNumberEditor	基本的にはプロパティエディタを設定する必要はない。
3	**プリミティブ型のラッパークラス**(boolean)	CustomBooleanEditor	基本的にはプロパティエディタを設定する必要はない。
4	Date 型	CustomDateEditor	必ずプロパティエディタを設定する必要がある。
5	BigDecimal 型	CustomNumberEditor	基本的にはプロパティエディタを設定する必要はない。
6	String	**なし**	プロパティエディタを設定する必要はない。
7	String[]	**なし**	プロパティエディタを設定する必要はない。 ※ただし、配列の利用は非推奨とする。 (Castor を利用する場合、データ件数が多いと性能が落ちるため。)
8	ArrayList	CustomCollectionEditor	必ずプロパティエディタを設定する必要がある。
9	JavaBean	**なし**	プロパティエディタを設定する必要はない。

※プロパティエディタの詳細は、それぞれの Javadoc を参照すること。通常使用しないプロパティエディタは省略している。

◆ エラーコード

➢ クエリ形式のリクエストデータの形式チェックエラーが発生した場合に、返却されるエラーコードの一覧を記す。

	エラーコード	置換文字列	概要
1	typeMismatch.byte	{属性名，データ型}	byte 型と異なるデータ型が入力された場合
2	typeMismatch.short	{属性名，データ型}	short 型と異なるデータ型が入力された場合

3	typeMismatch.int	{属性名，データ型}	int 型と異なるデータ型が入力された場合
4	typeMismatch.long	{属性名，データ型}	long 型と異なるデータ型が入力された場合
5	typeMismatch.short	{属性名，データ型}	short 型と異なるデータ型が入力された場合
6	typeMismatch.boolean	{属性名，データ型}	boolean 型と異なるデータ型が入力された場合
7	typeMismatch.java.lang.Byte	{属性名，データ型}	java.lang.Byte 型と異なるデータ型が入力された場合
8	typeMismatch.java.lang.Short	{属性名，データ型}	java.lang.Short 型と異なるデータ型が入力された場合
9	typeMismatch.java.lang.Integer	{属性名，データ型}	java.lang.Integer 型と異なるデータ型が入力された場合
10	typeMismatch.java.lang.Long	{属性名，データ型}	java.lang.Long 型と異なるデータ型が入力された場合
11	typeMismatch.java.lang.Float	{属性名，データ型}	java.lang.Float 型と異なるデータ型が入力された場合
12	typeMismatch.java.lang.Double	{属性名，データ型}	java.lang.Double 型と異なるデータ型が入力された場合
13	typeMismatch.java.math.BigDecimal	{属性名，データ型}	java.math.BigDecimal 型と異なるデータ型が入力された場合
14	typeMismatch.java.lang.Boolean	{属性名，データ型}	java.lang.Boolean 型と異なるデータ型が入力された場合
15	typeMismatch.java.util.Date	{属性名，データ型}	java.util.Date 型と異なるデータ型が入力された場合
16	typeMismatch	{属性名，データ型}	データ型と異なるデータ型が入力された場合

◆ 拡張ポイント

なし

■ 関連機能

- 『RB-03 XML-Object 変換機能』
- 『RD-01 例外ハンドリング機能』
- 『RF-01 形式チェック機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Rich 版) 機能網羅サンプル
  - UC102 リクエストデータ分析
    - jp.terasoluna.rich.functionsample.requestanalysis.*

- TERASOLUNA Server Framework for Java (Rich 版) チュートリアル

- 「2.3 単純なロジック」
- /webapps/WEB-INF/tutorial-servlet.xml
- /webapps/WEB-INF/tutorial-controller.xml 等

■ 備考

- なし

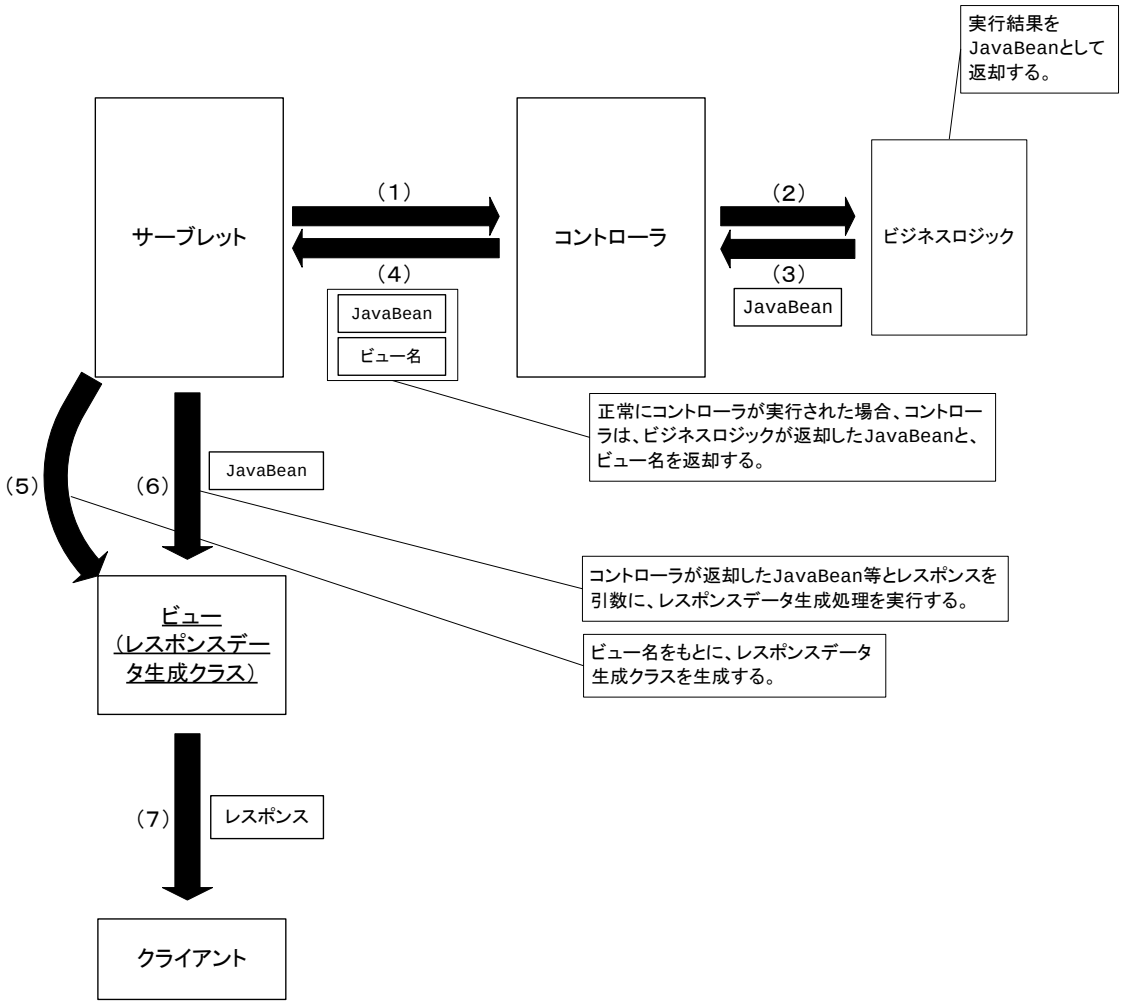
RB-02 レスポンスデータ生成機能

■ 概要

◆ 機能概要

- ビジネスロジックから返却された JavaBean をもとにレスポンスデータを生成し、クライアントに出力する機能である。
- レスポンスデータの形式としては、XML 形式とバイナリ形式を提供する。

◆ 概念図



◆ 解説

(1) サーブレット（ディスパッチャサーブレット）はコントローラを実行する。
(2) コントローラはビジネスロジックを実行する。
(3) ビジネスロジックは、実行結果をJavaBeanとして返却する。
(4) コントローラは、(3)で返却されたJavaBeanと、ビュー名を返却する。
(5) サーブレットは、ビュー名をもとに、ビュー（レスポンスデータ生成クラス）を生成する。ビューに関しての詳細は、後述の「TERASOLUNA Server Framework for Java (Rich版)が提供するビュー」を参照のこと。
コントローラから例外が返却された場合、例外のハンドリングを行う。
詳細は、『RD-01 例外ハンドリング機能』を参照のこと。
(6) サーブレットは、(4)で返却されたJavaBeanをもとに、ビューを実行する。

(7) ビューは、引数として渡されたJavaBeanからレスポンスデータを生成して、クライアントに出力する。

- TERASOLUNA Server Framework for Java (Rich 版)が提供するビュー
  TERASOLUNA Server Framework for Java (Rich 版)では、以下の３つのビューを用意している。
  ビューはリクエストごとに切り替えることが可能であり、用途に応じて使い分ける。
  - Castor ビュー
    XML 形式のレスポンスデータをクライアントに出力する際に使用するビュー。
    ビジネスロジックから返却された JavaBean を、マッピング定義に従い XML データに変換するか、マッピング定義を省略して FW で自動変換する 2 つの方法がある。
    詳細は『RB-03 XML-Object 変換機能』参照のこと。
    XML 形式のレスポンスデータを生成する場合、通常の場面では本ビューを使用することが推奨される。
    本ビューを使用するメリット・デメリットを以下に挙げる。
    - ✧ メリット
      - マッピング定義ファイルの形式が分かりやすく、修正も容易。
      - マッピング定義をもとに、確実に JavaBean を XML に変換することができる。
      - JavaBean と XML データの相互変換が可能。
        クライアントが Java の場合、送受信でマッピング定義ファイルを共用することができ、定義ファイルの量を減らすことができる。
      - マッピング定義を省略可能。
      - **電文に「xml:space=”preserve”」属性を付与すると通常では削除されてしまう空白を、削除せずに送信できる。**
        デフォルトで付与するようになっている。
        詳細は『RB-03 XML-Object 変換機能』参照のこと。
    - ✧ デメリット
      - Velocity ビューと比べると、パフォーマンスの面で劣る。
      - **電文に「xml:space=”preserve”」属性を付与すると XML レスポンスの整形が行えない。**
        デフォルトで付与するようになっている。
        詳細は『RB-03 XML-Object 変換機能』参照のこと。
  - Velocity ビュー
    XML 形式のレスポンスデータをクライアントに出力する際に使用するビュー。
    ビジネスロジックから返却された JavaBean を、テンプレートファイルを使用して、XML 形式のレスポンスデータに変換する。
    本ビューは電文生成時のパフォーマンスに優れ、巨大な XML 形式のレスポンスデータを生成する場合など、性能が要求される局面で使用されることが想定される。
    例外発生時のレスポンスデータは、本ビューを使用して生成される。詳細は、

『RD-01 例外ハンドリング機能』を参照のこと。
本ビューを使用するメリット・デメリットを以下に挙げる。

- ✧ メリット
  - テンプレートファイルから生成される電文が視覚的に分かりやすい。
  - Castor ビューと比べ、パフォーマンスの面で優れている。
  - XML 形式に限らず、CSV 形式、HTML 形式などのテキストデータを自由に生成することができる。
- ✧ デメリット
  - 生成される電文の妥当性が保障されないため、生成される XML データに対し妥当性検証を行う必要がある。

➢ ファイルダウンロードビュー
クライアントにファイルを送信する際に使用するビュー。
ビジネスロジックから返却された JavaBean をもとに、バイナリデータをレスポンスデータに設定する。

- ビューリゾルバ
  各ビューは、それぞれビューリゾルバ（ViewResolver 実装クラス）を持つ。
  ビューリゾルバは、与えられたビュー名を解決できる場合、ビュー名に対応するビューを返却する。ビュー名が解決できない場合は、null を返却する。
- ビューの優先度
  ViewResolver 実装クラスには優先度を設定することができる。
  - ➢ 優先度の変更を行うことは可能だが、デフォルトで用意されている３つのビューを使用する場合、特に変更する必要はない。
  - ➢ VelocityViewResolverEx には優先度を設定することができないので、検索される順番が必ず最後になる。

■ 使用方法

◆ コーディングポイント

- Castor ビューのコーディングポイント
  マッピング定義に従い、JavaBean を XML データに変換する。
  - 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
    Bean 定義の設定すべきポイントとして、下記の項目が挙げられる。
    1. CastorViewResolver
       Castor ビューを生成するためのビューリゾルバ。
       コントローラにビューの設定を行わない場合、デフォルトで Castor ビューが使用される。
       以下のプロパティを設定する。

	プロパティ名	必須	デフォルト値	概要
1	cache	×	true	ビューのキャッシュを行う場合は true。
2	requestContextAttribute	○	-	リクエストコンテキストの名前。 実際に Castor ビューでリクエストコンテキストをレスポンスデータ生成に使用する場合は、拡張を行う必要がある。
3	contentType	×	“text/html; charset=ISO-8859-1”	レスポンスの文字コード。
4	order	×	Integer.MAX_VALUE	ビューが使用される優先度。値の小さい方が早く使用される。 デフォルト値は Integer.MAX_VALUE のため、最後に使用される。同一の値が複数ある場合、ビューが使用される順番はランダムになるので、設定することを推奨する。
5	oxmapper	○	-	OXMapper 実装クラスを設定する。

    2. CastorOXMapperImpl
       Castor の機能を用い、JavaBean を XML に変換するためのクラス。
       詳細は『RB-03 XML-Object 変換機能』を参照のこと。
    3. コントローラの抽象定義
       Castor ビューを使用する各コントローラで、共通的に使用される親のコントローラの抽象 Bean 定義を有効化する。詳細は『RA-02 コントローラ拡張機能』を参照のこと。

✧ Bean 定義ファイルサンプル

```
<!-- CastorビューリゾルバのBean定義サンプル -->
<bean id="castorViewResolver"
    class="jp.terasoluna.fw.web.rich.springmvc.servlet.view.castor.
    CastorViewResolver">
  <property name="cache" value="true" />
  <property name="requestContextAttribute" value="rc" />
  <property name="contentType" value="text/xml;charset=UTF-8" />
  <property name="order" value="2" />
  <property name="oxmapper" ref="oxmapper"/>
</bean>

<!-- オブジェクト-XML変換クラスのBean定義サンプル -->
<bean id="oxmapper"
    class="jp.terasoluna.fw.oxm.mapper.castor.CastorOXMapperImpl">
    <property name="preserveWhitespaceAtMarshal" value="false" />
</bean>
```

ViewResolver 実装クラス。

ビューのキャッシュ。

リクエストコンテキストの名前。

レスポンスの文字コード。

ビューが使用される優先度。

OXMapper 実装クラスを設定する。

false にすることで xml-preserve 属性を付与しなくなる。
接頭接尾の空白・改行が保持できなくなるが、XML を整形できる。

➢ 業務開発者が行うコーディングポイント

ビジネスロジックから返却される JavaBean ごとに、テンプレートファイルを生成する必要がある。

✧ コントローラの Bean 定義

Castor ビューを使用するコントローラの Bean 定義を行う。

● Bean 定義サンプル

リクエストデータの形式が XML、ビジネスロジックが POJO、レスポンスデータの形式が XML であることを想定し、上記ソフトウェアアーキテクトのコーディングポイントで Bean 定義された pojoXmlRequestCastorViewController を親のコントローラに設定するサンプルを以下に記す。

```
<!-- コントローラのBean定義サンプル(受信リクエスト：XML形式 ビュー：Castor) -->
<bean name="/sumController"
    class="jp.terasoluna.sample2.web.controller.SumController"
    parent="pojoXmlRequestCastorViewController" scope="prototype">
  <property name="sumService" ref="sumService"/>
</bean>
```

親のコントローラを設定する。

✧ マッピング定義ファイルの作成

ビジネスロジックから返却される JavaBean に対応するマッピング定義ファイルを生成し、JavaBean と同じパスに配置する。

● マッピング定義ファイルのサンプル

ビジネスロジックから返却される JavaBean のサンプル“sample.dto.SampleDto”と、それに対応するマッピング定義ファイルを以下に記す。

詳細は、『RB-03 XML-Object 変換機能』を参照のこと。

➢ sample.dto.SampleDto（getter,setter,adder は省略）

```
package sample.dto
public class SampleDto {
  private int userid;
  private String username;
  private List<Item> item;
}
```

ビジネスロジックから返却される JavaBean。

JavaBean の配列。List 型や Map 型のマッピングも可能。

➢ sample.dto.Item（getter,setter,adder は省略）

```
package sample.dto
public class Item {
  private int id;
  private String name;
  private int price;
}
```

ビジネスロジックに格納される JavaBean。

- 上記 JavaBean サンプルに対応するマッピング定義ファイルサンプル
  ※マッピング定義を省略して FW で自動変換可能。
  詳細は『RB-03 XML-Object 変換機能』参照のこと。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Object Mapping DTD Version
1.0//EN"
"http://castor.exolab.org/mapping.dtd">
<mapping>
  <class name="sample.dto.SampleDto">
    <map-to xml="sample-dto"/>
    <field name="userid" type="int">
      <bind-xml name="user-id" node="element"/>
    </field>
    <field name="username" type="string">
      <bind-xml name="user-name" node="element"/>
    </field>
    <field name="item" type="sample.dto.Item" collection="arraylist"
/>

  </class>

  <class name="sample.dto.Item">
    <map-to xml="item"/>
    <field name="id" type="int">
      <bind-xml name="id" node="element"/>
    </field>
    <field name="name" type="string">
      <bind-xml name="name" node="element"/>
    </field>
    <field name="price" type="int">
      <bind-xml name="price" node="element"/>
    </field>
  </class>
</mapping>
```

SampleDto クラスの定義。

XML データの要素名を指定する。class 要素には map-to 要素、field 要素には bind-xml 要素で指定する。

繰り返し要素は配列で定義する。

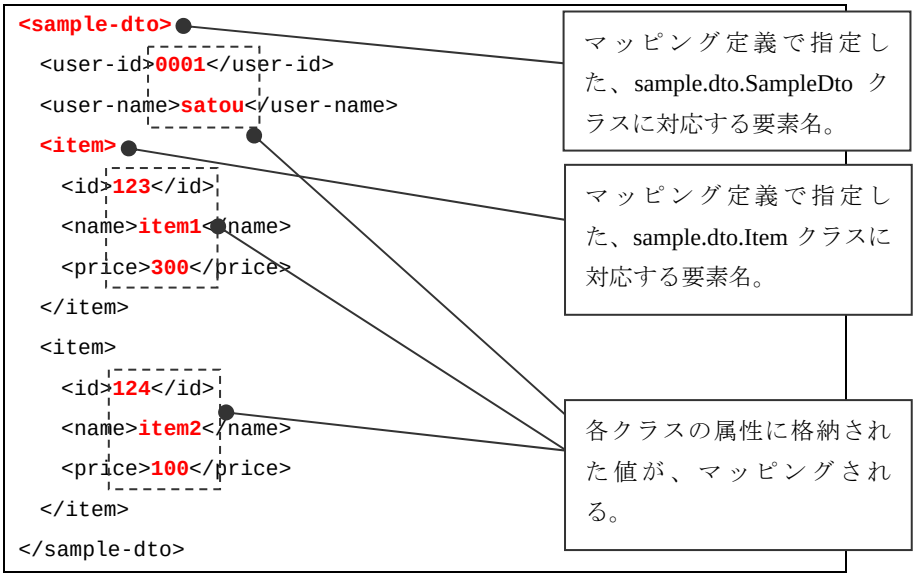
XML データの item 要素と sample.dto.Item クラスを関連付ける。

sample.dto.Item クラスの定義。

➢ 上記 JavaBean と、マッピング定義のサンプルから生成される XML
それぞれのクラスの属性に、以下の値が格納されていることとする。

	クラス	属性	値
1	sample.dto.SampleDto	userid	0001
2	sample.dto.SampleDto	username	satou
3	sample.dto.Item[0]	id	123
4	sample.dto.Item[0]	name	item1
5	sample.dto.Item[0]	price	300
6	sample.dto.Item[1]	id	124
7	sample.dto.Item[1]	name	item2
8	sample.dto.Item[1]	price	100

➢ 生成される XML 形式のレスポンスデータ



※ マッピング定義を省略してFWで自動変換する場合、Bean名からXML要素名を生成できる。
詳細は『RB-03 XML-Object変換機能』参照のこと。

- Velocityビュー
  - 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
    以下の項目のBean定義を行う。
    1. VelocityViewResolverEx
       Velocity ビューを生成するための VelocityViewResolver 拡張クラス。
       以下のプロパティを設定する。

	プロパティ名	必須	デフォルト値	概要
1	cache	×	true	ビューのキャッシュを行う場合は true。
2	requestContextAttribute	○	-	ここで設定した名前を通じて、テンプレートファイルから、ビューが生成したリクエストコンテキストにアクセスすることができる。 例）requestContextAttribute に文字列“rc”を設定した場合、テンプレートファイルにおいて、変数 error に対応するエラーメッセージを取得する場合の記述。 「 ${rc.getMessage($error)} 」
3	prefix	×	空文字	テンプレートファイルの接頭辞。
4	suffix	×	空文字	テンプレートファイルの接尾辞。
5	encoding	×	“ISO-8859-1”	テンプレートファイルの文字コード。
6	contentType	×	“text/html; charset=ISO-8859-1”	レスポンスヘッダに設定される Content-Type（文字コード）。
7	exposeSpringMacroHelpers	×	false	VelocityTool など、マクロを使用する場合は true。
8	toolboxConfigLocation	×		VelocityTool を使用するときの設定ファイルの場所。

    2. VelocityConfigurer
       Velocity の環境をセットアップするクラス。
       テンプレートファイルを配置するパスを設定する。
       ※テンプレートファイルの名前は、「VelocityViewResolverEx の prefix 属性の値 ＋ リクエスト名 ＋ VelocityViewResolverEx の suffix 属性の値」の式で求められる。
    3. コントローラの抽象定義
       Velocity ビューを使用する各コントローラで、共通的に使用される親コントローラの抽象 Bean 定義を有効化する。詳細は『RA-02 コントローラ拡張機能』を参照のこと。

✧ Bean 定義ファイルサンプル

```
<!--
テンプレートファイルが配置されているルートパスに基づいて、
Velocity環境をセットアップするBean定義サンプル。
-->
<bean id="velocityConfig"
    class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
  <property name="resourceLoaderPath" value="/WEB-INF/velocity/" />
</bean>


<!-- VelecityビューリゾルバのBean定義サンプル -->
<bean id="velocityViewResolver"
    class="jp.terasoluna.fw.web.rich.springmvc.servlet.view.velocity.
    VelocityViewResolverEx">
  <property name="requestContextAttribute" value="rc" />
  <property name="prefix" value="" />
  <property name="suffix" value=".vm" />
  <property name="encoding" value="UTF-8" />
  <property name="contentType" value="text/xml;charset=UTF-8" />
  <property name="exposeSpringMacroHelpers" value="true"/>
  <property name="toolboxConfigLocation" value="/WEB-INF/velocity/toolbox.xml"/>
</bean>
```

Velocity 環境を
セットアップするクラス。

ここで定義したパス配下に、生成した
Velocity テンプレートファイルを配置する。

Velocity ビューを生成するための ViewResolver 実装クラス。

リクエストコンテキストの名前。

接頭辞。

接尾辞。

レスポンスの文字コード。

マクロの使用。

VelocityTool の設定ファイル

テンプレートファイルの文字コード。

- 業務開発者が行うコーディングポイント
  - ✧ コントローラの Bean 定義

    親のコントローラとして、ソフトウェアアーキテクトが抽象定義した、Velocity ビューを使用するリクエストコントローラを設定する。
    - Bean 定義サンプル

      リクエストデータの形式が XML、ビジネスロジックが POJO、レスポンスデータの形式が XML であることを想定し、上記ソフトウェアアーキテクトのコーディングポイントで Bean 定義された pojoXmlRequestVelocityViewController を親のコントローラに設定するサンプルを以下に記す。

```
<!-- コントローラのBean定義サンプル
(受信リクエスト：XML形式 ビュー：Velocity)
-->
<bean name="/sumController"
   class="jp.terasoluna.sample2.web.controller.SumController"
   parent="pojoXmlRequestVelocityViewController" scope="prototype">
  <property name="sumService" ref="sumService"/>
</bean>
```

親のコントローラを設定する。

    - Velocity テンプレートファイルの生成

      各コントローラに対応するテンプレートファイルを生成し、上記ソフトウェアアーキテクトの Bean 定義より指定されたパスに配置する。

      jp.terasoluna.fw.web.rich.springmvc.Constants クラスの ERRORCODE_KEY 属性に設定された値（デフォルトは“ret”）を通じて、ビジネスロジックから返却された JavaBean にアクセスすることができる。例えば、ERRORCODE_KEY の値が“ret”の場合、変数 ret で、ビジネスロジックから返却された JavaBean にアクセスすることができる。

      VelocityViewResolverEx の requestContextAttribute 属性に設定された値を通じて、ビューが生成したリクエストコンテキストにアクセスすることができる（上記ソフトウェアアーキテクトの Velecity ビューリゾルバの Bean 定義サンプル参照のこと）。requestContextAttribute の値が“rc”の場合、変数 rc でリクエストコンテキストにアクセスできる。

      ※リクエストコンテキストを使用することで、メッセージリソースや例外オブジェクトへのアクセスが容易になる。

      メッセージリソースの取得方法は、『CE-01 メッセージ管理機能』を参照のこと。

➢ テンプレートファイルサンプル

以下に、ビジネスロジックが返却する JavaBean の属性 result に数値 300 が、属性 code に文字列“sample”が格納されている場合についてのテンプレートファイルのサンプルを記す。

リクエスト名が“sum”、VelocityViewResolverEx の prefix 属性の値に空文字、VelocityViewResolverEx の suffix 属性の値に“.vm”、requestContextAttribute 属性の値に“rc”が設定されていることとする。

この場合、ファイル名は“sum.vm”となる。（「prefix＋リクエスト名＋suffix」）

3 行目で、属性 result の値を表示する。

4 行目で、リクエストコンテキスト（変数“rc”）の getMessage メソッドを実行し、メッセージリソースから“message.sample”のキーに対応する値を取得して表示する。（“message.”と属性 code（“sample”）の値を文字列連結してキーとする）

✧ sum.vm

```
<?xml version="1.0" encoding="utf-8" ?>
<body>
  Ans. ${ret.result}<br>
  Msg. ${rc.getMessage("message.${ret.code}")}
</body>
```

変数 ret に、ビジネスロジックから返却された JavaBean が格納されている。

プロパティファイルに「“message”+JavaBean の属性 code の値」のキーで格納されたメッセージを取得す

✧ プロパティファイルサンプル（errors.properties.sjis）

```
message.sample=サンプル用のメッセージ
```

✧ 上記サンプル（sum.vm）から生成されるレスポンスデータ

```
<?xml version="1.0" encoding="utf-8" ?>
<body>
  Ans. 300<br>
  Msg. サンプル用のメッセージ
</body>
```

属性 result に格納されていた数値 300 に置換される。

“message.sample”のキーで取得されたメッセージに置換される。

✧　特殊文字のサニタイジング（無害化）

Velocity ビューを使用して XML 電文を生成する場合、xml 形式の特殊文字を以下のようにエスケープする必要がある。

- < ⇒ <
- “ ⇒ "
- & ⇒ &
-]]> ⇒]]>

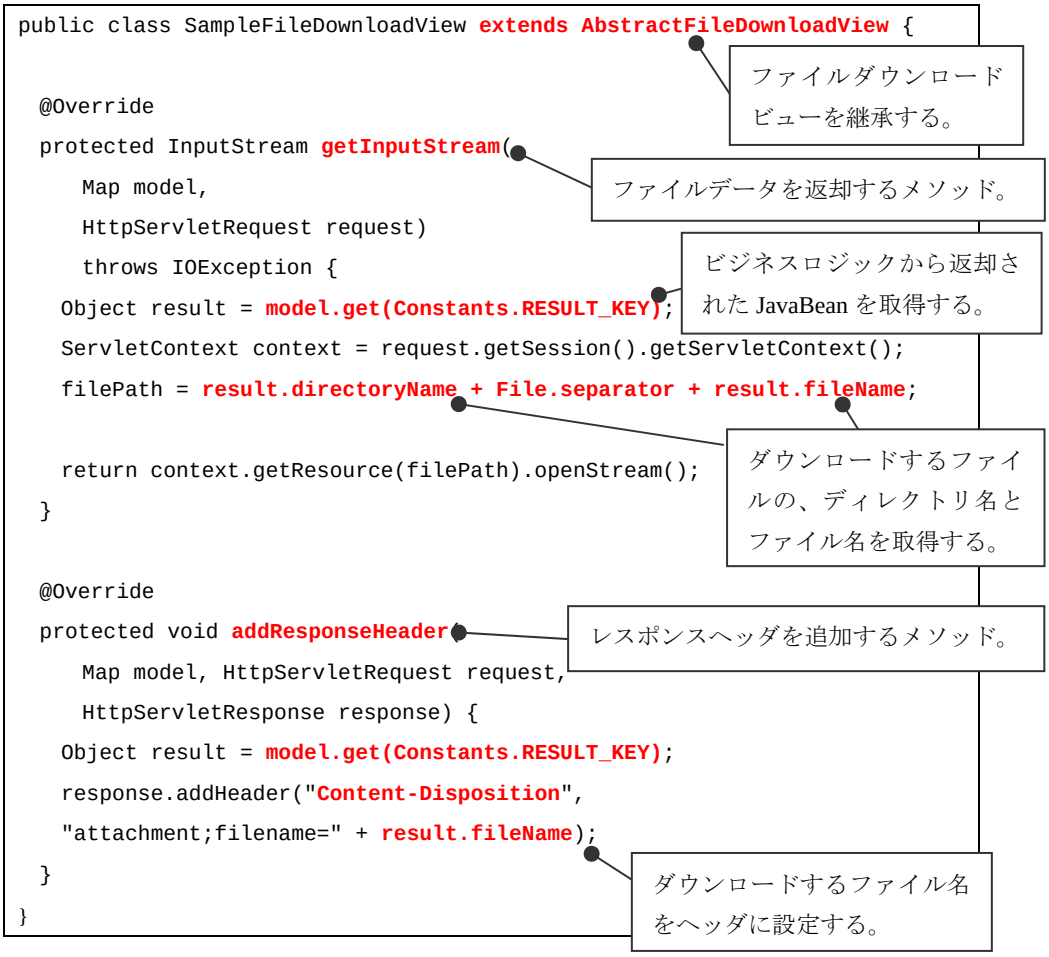
レスポンスデータに上記の特殊文字が入る可能性がある場合、
TERASOLUNA Server Framework for Java (Rich 版)では VelocityTool を使用してエスケープを行う。
記述方法は以下の通りである。

```
$esc.xml(エスケープ対象の文字列)
```

以下はテンプレートファイルの例である。

```
<?xml version="1.0" encoding="utf-8" ?>
<body>
   ********** Velocity Sample **********<br>

   Ans. $esc.xml(${ret.result})
</body>
```

- ファイルダウンロードビュー
  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
  - AbstractFileDownloadView 継承クラスの生成
    TERASOLUNA Server Framework for Java (Rich 版)が提供するファイルダウンロードビュー（AbstractFileDownloadView）は、抽象クラスであるため、以下のメソッドを、サブクラスで実装する必要がある。
    1. ダウンロードするファイル情報（入力ストリーム）を取得するメソッド
    2. レスポンスヘッダを追加するメソッド
       - ビジネスロジックから返却された JavaBean は、引数の model オブジェクトから、Constants.RESULT_KEY をキーとして取得することができる。
       - ファイルダウンロードビューのサブクラスサンプル
         ビジネスロジックから返却された JavaBean の属性 fileName に格納されたファイル名をもとに、ファイルのダウンロードを行う機能を持ったサブクラスのサンプルを以下に記す。
         例えば、ビジネスロジックから返却された JavaBean の directoryName 属性が“/tmp”、fileName 属性が“sample.pdf”の場合、“/tmp/”フォルダ配下に配置された “sample.pdf”という名前のファイルがダウンロードされる。
         - サブクラスサンプル（SampleFileDownloadView.java）



- ResourceBundleViewResolver の Bean 定義

1. ResourceBundleViewResolver
   Velocity ビューを生成するために、使用するビューリゾルバ。
   ResourceBundleViewResolver はプロパティファイル等のメッセージリソースを使用して、ビュー名と使用するビューを対応付ける。
   メッセージリソースに「ビュー名＝ビュークラス名」で定義する。
   以下のプロパティを設定する。

	プロパティ名	必須	デフォルト値	概要
1	basename	×	“views”	ビュー名とビュークラスの関連が記述されたメッセージリソースの名前を設定する。 例）basename の値が“views”の場合、views.properties から、ビュー名とビュークラスの関連情報を取得する。
2	order	×	Integer.MAX_VALUE	ビューが使用される優先度。 デフォルト値は Integer.MAX_VALUE のため、最後に使用される。同一の値が複数ある場合、ビューが使用される

				順番はランダムになるので、設定することを推奨する。

2. コントローラの抽象定義
   ファイルダウンロードビューを使用する各コントローラで、共通的に使用する親コントローラを抽象定義する。
   viewName 属性に、メッセージリソースに定義したビュー名を設定する。

✧ Bean 定義サンプル
リクエストデータの形式がクエリ形式、ビジネスロジックが POJO、レスポンスデータの形式がバイナリであるコントローラから継承されることを想定した、親コントローラを抽象定義するサンプルを以下に記す。

```
<!-- ファイルダウンロードビューリゾルバのBean定義サンプル-->
<bean id="fileDownloadViewResolver"
    class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
  <property name="basename" value="views"/>
  <property name="order" value="1" />
</bean>


<!-- コントローラの抽象Bean定義サンプル
（受信リクエスト：クエリ形式、ビジネスロジック：POJO -->
<bean id="pojoXmlRequestDefaultFileDownloadViewController"
    abstract="true" parent="pojoXmlRequestController">
    <property name="viewName" value="FileDownloadSample" />
</bean>
```

取得するメッセージリソースの名前。

ビューが使用される優先順位。

XML 形式のリクエストデータをバインドするコントローラの抽象定義。詳細は『RB-01 リクエストデータ解析機能』を参照のこと。

✧ プロパティファイルサンプル（views.properties）

```
## この設定ファイルでビュー名と使用するビューを対応付ける。
## 書式は、<ビュー名>.class=<使用するビュー>となる。
## (.classを付加するのはSpringフレームワークの仕様。)
FileDownloadSample.class=jp.terasoluna.rich.functionsample.response.vie
w.FileResponseView
```

ビュー名と使用するビューを対応付ける。

➢ 業務開発者が行うコーディングポイント

✧ コントローラの Bean 定義

親のコントローラとして、ソフトウェアアーキテクトが抽象定義した、ファイルダウンロードビューを使用するリクエストコントローラを設定する。

● Bean 定義サンプル

リクエストデータの形式がクエリ形式、ビジネスロジックが POJO、レスポンスデータの形式がバイナリであるコントローラを想定し、上記ソフトウェアアーキテクトのコーディングポイントで Bean 定義された pojoQueryRequestVelocityViewController を親のコントローラに設定するサンプルを以下に記す。

```
<!-- コントローラのBean定義サンプル -->
<bean name="/velocityResponseController"

class="jp.teraoluna.rich.functionsample.response.controller.ResponseControll
er"
  parent="pojoQueryRequestVelocityViewController" scope="prototype">
    <property name="responseService" ref="responseService"/>
</bean>
```

■ リファレンス

◆ 構成クラス

	クラス名	概要
1	org.springframework.web.servlet.View	Spring が提供する、HTTP レスポンスデータを生成するクラスが実装すべきインタフェース
2	org.springframework.web.servlet.ViewResolver	ビュー名に対応するビューを返すクラスが実装すべきインタフェース。
3	jp.terasoluna.fw.web.rich.springmvc.servlet.view.castor.CastorView	Castor を利用して HTTP レスポンス生成を行うクラス。
4	jp.terasoluna.fw.web.rich.springmvc.servlet.view.castor.CastorViewResolver	Castor ビューを生成するための ViewResolver 実装クラス。 ビュー名が空文字、または Null の場合、Castor ビューを使用する。
5	org.springframework.web.servlet.view.velocity.VelocityView	Spring が提供する、Velocity を利用して HTTP レスポンス生成を行うクラス。
6	org.springframework.web.servlet.view.velocity.VelocityViewResolver	Spring が提供する、Velocity ビューを生成するための ViewResolver 実装クラス。
7	jp.terasoluna.fw.web.rich.springmvc.servlet.view.filedownload.AbstractFileDownloadView	バイナリファイルをダウンロードする際に利用する View 抽象クラス。 本クラスを継承したビューを生成するために、ResourceBundleViewResolver を使用する。
8	org.springframework.web.servlet.view.ResourceBundleViewResolver	Spring が提供する、プロパティファイルの定義をもとに、ビューを生成するための ViewResolver 実装クラス。
9	org.springframework.web.servlet.DispatcherServlet	Spring が提供する、一連の処理フローを実行するサーブレット。 コントローラから返却されたビュー名と対応するビューを生成し、レスポンスデータ生成処理をビューに委譲する。
10	org.springframework.web.servlet.support.RequestContext	Spring が提供するコンテキストクラス。本クラスを使用することで、メッセージリソースや例外オブジェクトへ容易にアクセスすることが可能となる。

◆ 拡張ポイント

(1) レスポンスヘッダの追加（CastorViewを使う場合）

以下の手順に従った拡張を行うことで、アプリケーションで共通的に付与したいレスポンスヘッダを追加することができる。
（サンプルでは、HTTP レスポンスに名称「header01」、値「abc」のレスポンスヘッダを追加している）

手順１：CastorViewクラスを継承したクラスを作成する。addResponseHeaderメソッドをオーバーライドし、HTTPレスポンスのヘッダ追加処理を実装する。

CastorView を継承して作成した拡張 View クラスのサンプル（CastorViewEx.java）

```
package sample.view;
  (import 文は省略)
public class CastorViewEx extends CastorView {
    /**
     * レスポンスヘッダを追加する。
     * レスポンスヘッダを追加する場合、このメソッドをオーバライドする。
     *
     * @param model 業務処理の結果
     * @param request HTTPリクエスト
     * @param response HTTPレスポンス
     */
     @Override
     protected void addResponseHeader(Map model,
              HttpServletRequest request, HttpServletResponse response) {
        response.addHeader("header01", "abc");
     }
}
```

CastorView クラスを継承する

addResponseHeader メソッドをオーバーライドする

レスポンスヘッダの追加処理を実装する

手順２：Bean 定義ファイルの CastorViewResolver 定義を以下のように修正し、手順１で作成した拡張 View クラスの完全修飾名を viewClass プロパティに記述する。

Bean 定義ファイルのサンプル（/webapps/WEB-INF/blank-servlet.xml）

```
･･･（中略）･･･
<!-- Castor用ビューリゾルバ -->
  <bean id="castorViewResolver"
      class="jp.terasoluna.fw.web.rich.springmvc.servlet.view.castor.CastorViewResolver">
    <property name="cache" value="true"/>
    <property name="requestContextAttribute" value="rc"/>
    <property name="contentType" value="text/xml;charset=UTF-8"/>
    <property name="order" value="2"/>
    <property name="oxmapper" ref="oxmapper"/>
    <property name="viewClass" value="sample.view.CastorViewEx"/>
  </bean>
･･･（中略）･･･
```

viewClass プロパティを追加する

■ 関連機能

- 『RA-02 コントローラ拡張機能』
- 『RB-03 XML-Object 変換機能』
- 『RD-01 例外ハンドリング機能』
- 『RC-01 ビジネスロジック実行機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Rich 版) 機能網羅サンプル
  - UC103 レスポンスデータ生成
    - ✧ jp.terasoluna.rich.functionsample.response.*

- TERASOLUNA Server Framework for Java (Rich 版) チュートリアル
  - 「2.3 単純なロジック」
  - /webapps/WEB-INF/tutorial-servlet.xml
  - /webapps/WEB-INF/tutorial-controller.xml
  - /webapps/WEB-INF/velocity/* 等

■ 備考

- なし

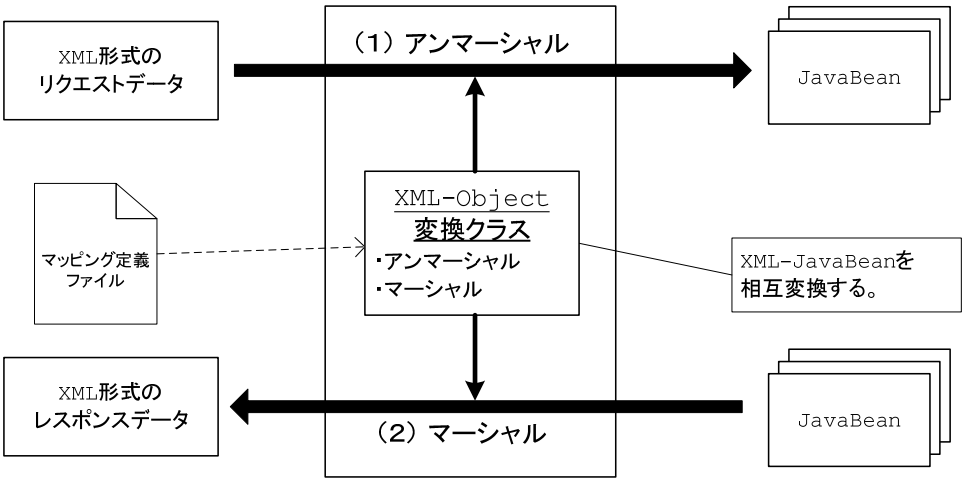
RB-03 XML-Object変換機能

■ 概要

◆ 機能概要

- XML と JavaBean の相互変換を行う機能である。
  TERASOLUNA Server Framework for Java (Rich 版)において、XML 形式のリクエストデータから JavaBean、及び、JavaBean から XML 形式のレスポンスデータへの変換を行う。
  - TERASOLUNA Server Framework for Java (Rich 版)のデフォルト実装では、Castor の機能を用いて変換を行う。

◆ 概念図



◆ 解説

1. マッピング定義を記述する場合
   - XML-Object変換クラスは、マッピング定義に従い、XML形式のリクエストデータを対象のJavaBeanに変換する。（アンマーシャル）
     『RB-01 リクエストデータ解析機能』で使用される。
   - XML-Object変換クラスは、マッピング定義に従い、JavaBeanをXML形式のレスポンスデータに変換する。（マーシャル）
     『RB-02 レスポンスデータ生成機能』で使用される。
2. マッピング定義を記述しない場合
   - XML-Object 変換クラスは、TERASOLUNA Server Framework for Java (Rich 版)が XML 形式のリクエストデータを対象の JavaBean に自動変換する。（アンマーシャル）

- XML-Object 変換クラスは、JavaBean を XML 形式のレスポンスデータに自動変換する。（マーシャル）

■ 使用方法

◆ コーディングポイント

- XML-Object 変換クラスの Bean 定義
  XML 形式のリクエストデータをバインドする場合、Bean 定義ファイルに XML-Object 変換クラスの設定を行う。
  - ✧ 詳細は、『RB-01 リクエストデータ解析機能』を参照のこと。
- XML データと JavaBean のマッピング定義（オプション）
  XML データと JavaBean の相互変換を行う場合、マッピング定義ファイルに設定すべきポイントとして、下記の項目が挙げられる。
  1. XML データと JavaBean のマッピングを定義する。
  2. マッピング定義ファイルのファイル名を、リクエストデータのバインド先の JavaBean の“クラス名 + .xml”にして、同じディレクトリに配置する。
     例）JavaBean が「sample.SampleDto2」クラスの場合、スキーマ定義ファイルはクラスパス上の「sample/SampleDto2.xml」ファイルとなる。
  3. 名前空間を宣言する。（オプション）
     スキーマ定義ファイルで名前空間を宣言した場合は、マッピング定義ファイルでも名前空間を宣言する。スキーマ定義ファイルに関しての詳細は、『RF-01 形式チェック機能』を参照すること。

- 名前空間を宣言しない場合のサンプル
  - XML データサンプル

```
<SampleDto2>
  <userId>25</userId>
  <name>テラソルナユーザ</name>
  <item>
     <id>1</id>
     <name>item1</name>
     <price>1000</price>
  </item>
  <item>
     <id>2</id>
     <name>item2</name>
     <price>2000</price>
  </item>
  <member>true</member>
  <registorDate>2007-06-01T12:34:56</registorDate>
</SampleDto2>
```

ルート要素に 1 つの JavaBean を対応させる

子要素が存在する要素に 1 つの JavaBean を対応させる。

  - マッピング対象の JavaBean サンプル（getter,setter,adder は省略）

```
public class SampleDto2 {
  private int userId;
  private String name;
  private  List<Item2> itemList = new ArrayList<Item2>();
  private boolean member;
  private Date registorDate;
}

public class Item2 {
  private int id;
  private String name;
  private int price;
}
```

繰り返し要素は List で定義する。

✧ マッピング定義ファイルサンプル

※オブジェクトを XML に変換する場合、マッピング定義された順番に XML の要素が出力される。

例えば、下記コードの場合、「<userId> → <name> → <item>」の順番で、

✧ XML の要素が出力される。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Object Mapping DTD Version
1.0//EN" "http://castor.exolab.org/mapping.dtd">
<mapping>
  <class name="jp.terasoluna.rich.sample.service.bean.SampleDto2">
    <map-to xml="SampleDto2" />
    <field name="userId" type="java.lang.Integer">
      <bind-xml name="userId" node="element" />
    </field>
    <field name="name" type="java.lang.String">
      <bind-xml name="name" node="element" />
    </field>
    <field name="item"
           type="jp.terasoluna.rich.sample.service.bean.Item2"
           collection="arraylist">
      <bind-xml name="item" node="element" />
    </field>
    <field name="member" type="java.lang.Boolean">
      <bind-xml name="member" node="element" />
    </field>
    <field name="registorDate" type="java.util.Date">
      <bind-xml name="registorDate" node="element" />
    </field>
  </class>

  <class name="jp.terasoluna.rich.sample.service.bean.Item2">
    <field name="id" type="java.lang.Integer">
      <bind-xml name="id" node="element" />
    </field>
    <field name="name" type="java.lang.String">
      <bind-xml name="name" node="element" />
    </field>
    <field name="price" type="java.lang.Integer">
      <bind-xml name="price" node="element" />
    </field>
  </class>
</mapping>
```

SampleDto2 クラスの定義。

XML データの SampleDto2 要素と JavaBean を関連付ける。

List 型要素は arraylist で定義する。

XML データの item 要素と JavaBean を関連付ける。

Item2 クラスの定義。

- 名前空間を宣言する場合のサンプル
  - XML データサンプル

```
<SampleDto2 xmlns="http://xxx.co.jp/sample/uselistrequest2">
  <userId>25</userId>
  <name>テラソルナユーザ</name>
  <item>
     <id>1</id>
     <name>item1</name>
     <price>1000</price>
  </item>
  <item>
     <id>2</id>
     <name>item2</name>
     <price>2000</price>
  </item>
  <member>true</member>
  <registorDate>2007-06-01T12:34:56</registorDate>
</SampleDto2>
```

一意の名前空間の宣言。（デフォルト名前空間）

ルート要素に 1 つの JavaBean を対応させる。

子要素が存在する要素に 1 つの JavaBean を対応させる。

  - マッピング対象の JavaBean サンプル（getter,setter,adder は省略）

```
public class SampleDto2 {
  private int userId;
  private String name;
  private  List<Item2> itemList = new ArrayList<Item2>();
  private boolean member;
  private Date registorDate;
}

public class Item2 {
  private int id;
  private String name;
  private int price;
}
```

繰り返し要素は List で定義する。

✧ マッピング定義ファイルサンプル

※オブジェクトを XML に変換する場合、マッピング定義された順番に XML の要素が出力される。

例えば、下記コードの場合、「<userId> → <name> → <item>」の順番で、XML の要素が出力される。

```
<?xml version="1.0" encoding="UTF-8"?>
<mapping>
  <class name="jp.terasoluna.rich.sample.service.bean.SampleDto2">
    <map-to xml="SampleDto2" xmlns="http://xxx.co.jp/sample/uselistrequest2"
            ns-uri="http://xxx.co.jp/sample/uselistrequest2" />
    <field name="userId" type="java.lang.Integer">
      <bind-xml name="userId" node="element" />
    </field>
    <field name="name" type="java.lang.String">
      <bind-xml name="name" node="element" />
    </field>
    <field name="item" type="jp.terasoluna.rich.sample.service.bean.Item2"
          collection="arraylist">
      <bind-xml name="item" node="element" />
    </field>
    <field name="member" type="java.lang.Boolean">
      <bind-xml name="member" node="element" />
    </field>
    <field name="registorDate" type="java.util.Date">
      <bind-xml name="registorDate" node="element" />
    </field>
  </class>

  <class name="jp.terasoluna.rich.sample.service.bean.Item2">
    <map-to xml="item"
            ns-uri="http://xxx.co.jp/sample/uselistrequest2" />
    <field name="id" type="java.lang.Integer">
      <bind-xml name="id" node="element" />
    </field>
    <field name="name" type="java.lang.String">
      <bind-xml name="name" node="element" />
    </field>
    <field name="price" type="java.lang.Integer">
      <bind-xml name="price" node="element" />
    </field>
  </class>
</mapping>
```

名前空間の宣言。

SampleDto2 クラスの定義。

名前空間の宣言。

XML データの SampleDto2 要素と JavaBean を関連付ける。

繰り返し要素は arraylist で定義する。

XML データの item 要素と JavaBean を関連付ける。

Item2 クラスの定義。

名前空間の宣言。

- マッピング定義を省略し FW より XML データと JavaBean を自動変換させる。
  - TERASOLUNA Server Framework for Java (Rich 版)の「XML-Object 変換機能」を利用する場合、JavaBean のマッピング定義を省略可能である。
    - DTO クラスの Collection 継承クラスを使用する場合は Bean 定義に setter,getter の他に adder が必要である。
    - Map は正常に Unmarshaller されないため、使用しない。
    - デフォルトの変換ルールは以下の通りとなっている。
      - Unmarshaller
        入力 Bean として指定したクラスがルートノードとして使用される。
        例)<inputBean> ⇒ InputBean クラス
      - Marshaller
        Bean 名から XML 要素名を命名する。
        例)InputBean クラス ⇒ <inputBean>

      Unmarshaller と Marshaller の整合性をとるために変換ルールを変更する必要がある。そのため、プロパティファイルと DTO クラスを以下のように設定する。

- プロパティ（castor.properties）設定例

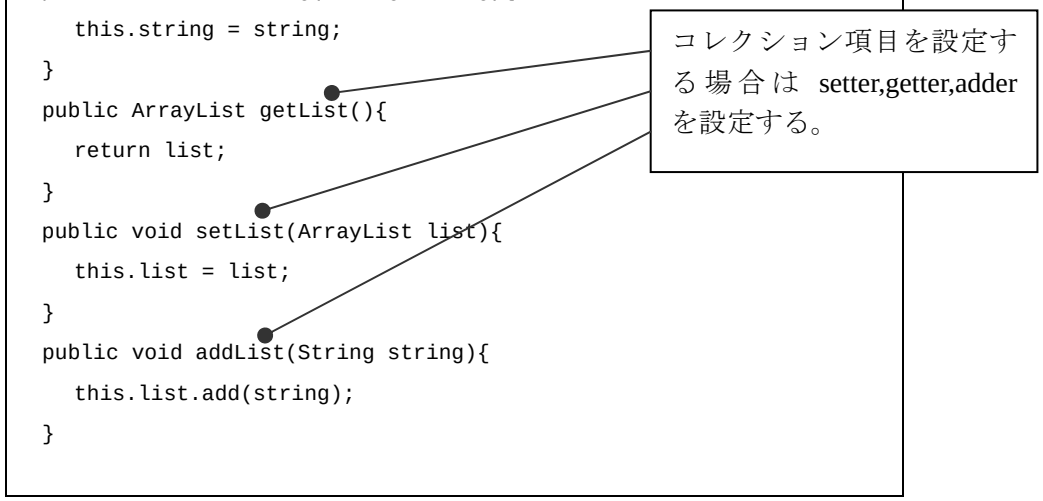
```
#
# Castorプロパティファイル
#
# クラスパス上にこのファイルを置くと、Castorが実行時に
# このファイルのプロパティを読み込む
#
org.exolab.castor.indent=true
org.exolab.castor.xml.saveMapKeys=false
org.exolab.castor.xml.naming=mixed
org.exolab.castor.xml.introspector.primitive.nodetype=element
```

デフォルトの XML 要素ネーミングルールの変更。
例） InputBean クラス ⇒ <inputBean>となる。

プリミティブ項目を要素として Marshal する設定

- DTO クラス設定

```
/**
 * DTOクラス。
**/
public class SampleDTO implements Serializable {
  private String string = null;
  // コレクション項目
  private List<String> list = new ArrayList<String>();
  public String getString(){
     return string;
  }
  public void setString(String string){
     this.string = string;
  }
  public ArrayList getList(){
     return list;
  }
  public void setList(ArrayList list){
     this.list = list;
  }
  public void addList(String string){
     this.list.add(string);
  }
```



✧ スキーマバリデータを使用する場合は以下の定義が必要となる

- Bean 定義を以下のように設定する。

```
<?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"
      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">
                                          :
                                        省略
                                          :
<bean id=”schemaValidator”
      class=”jp.terasoluna.fw.oxm.xsd.xerces.SchemaValidatorImpl” >
      <property name=”namespace” value=”true”/>
</bean>
                                          :
                                        省略
                                          :
```

true に設定した場合(デフォルトは false)、電文にネームスペースが付与されていないと例外が発生する。

- XML を整形したい場合
  Castor マッピング無設定時に XML を整形したい場合は以下の設定を行う。
  - サーバ側の場合
    Bean 定義に以下の設定を行う。

```
<?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"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.50.xsd
http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util-2.5.xsd">
                                        :
                                    省略
                                        :
<bean id="oxmapper"
   class="jp.terasoluna.fw.oxm.mapper.castor.CastorOXMapperImpl">
     <property name="preserveWhitespaceAtMarshal" value="false"/>
  </bean>
                                        :
                                    省略
                                        :
```

- value=”true”(デフォルト)
「xml:space="preserve"」がルートノードに付与される。その際 XML レスポンスが整形されなくなる。
接頭接尾の空白・改行を保持したい場合は true(デフォルト)にする。
- value=”false”
XML レスポンスが整形される。
ただし、「xml:space="preserve"」がルートノードに付与されない。

- クライアント側からサーバ側に電文を送る時に無条件に接頭接尾の空白・改行を保持したい場合。

  - Bean 定義に以下の設定を行う。

```
<?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"
      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">
                                        :
                                    省略
                                        :
<bean id="oxmapper"
   class="jp.terasoluna.fw.oxm.mapper.castor.CastorOXMapperImpl">
     <property name="preserveWhitespaceAtUnmarshal" value="true"/>
  </bean>
                                        :
                                    省略
                                        :
```

false(デフォルト)になっても XML リクエストに <xml:space=”preserve”>が付与されていれば接頭接尾の空白・改行を保持する。

■ リファレンス

◆ 構成クラス

	クラス名	概要
1	OXMapper	Object-XML 変換を行うためのインタフェース。
2	CastorOXMapperImpl	Castor の機能を利用して XML-Object 変換を行う OXMapper デフォルト実装クラス。

◆ Castorがサポートするデータ型

- <field>要素の type 属性に記述する。

	名前	データ型
1	string	java.lang.String
2	int	int or java.lang.Integer
3	long	long or java.lang.Long
4	Boolean	boolean or java.lang.Boolean
5	double	double or java.lang.Double
6	float	float or java.lang.Float
7	big-decimal	java.math.BigDecimal（整数のみ使用可能。小数点は使用不可）
8	byte	byte or java.lang.Byte
9	date	java.util.Date
10	short	short or java.lang.Short
11	char	char or java.lang.Character

※通常使用しない定義は省略している。詳細は Castor のドキュメントを参照すること。

◆ Castorがサポートするコレクション型

- 配列型の属性にマッピングする際に使用する。
  <field>要素の collection 属性に記述する。

	名前	配列型
1	※array (非推奨)	配列
2	arraylist	java.util.Arraylist ， java.util.List
3	collection	java.util.Collection ， java.util.Arraylist

※データ件数が多い場合、配列型のプロパティを利用すると性能が落ちるため、非推奨とする。

※通常使用しない定義は省略している。詳細は Castor のドキュメントを参照すること。

◆ 拡張ポイント

- CastorOXMapperImpl クラスの getUrl()メソッドをオーバーライドすることで、マッピング定義ファイルの取得方法を変更することができる。

■ 関連機能

- 『RB-01 リクエストデータ解析機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Rich 版) チュートリアル
  - 「2.3 単純なロジック」
  - jp.terasoluna.rich.tutorial.service.bean.UserBean.java

- TERASOLUNA Server Framework for Java (Rich 版) **機能網羅サンプル**
  - jp.terasoluna.rich.functionsample.response.bean.FileResponseData.java
  - jp.terasoluna.rich.functionsample.response.bean. FileResponseData[※].xml

  ※上記のファイルを確認する場合は以下の処理を実行する。
  「ant/build.xml」を右クリック「実行」→「Ant ビルド. . . 」を選択する。
  「ターゲット」タブを選択し、「useCastorMappingFile」にチェックをいれ実行する。

■ 備考

- なし

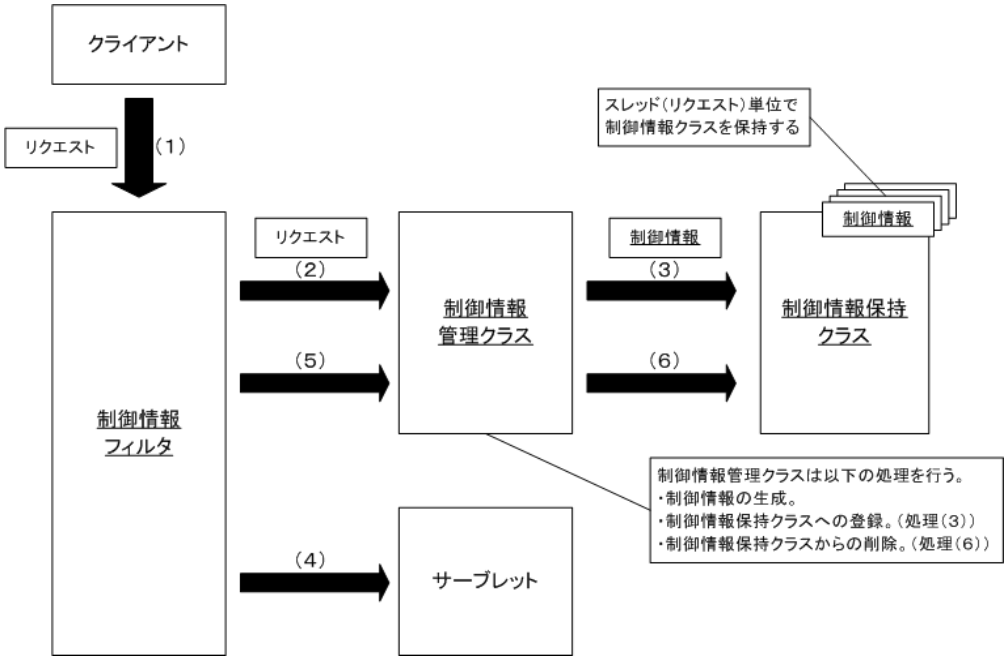
RB-04 制御情報管理機能

■ 概要

◆ 機能概要

- リクエスト処理で使用する共通の制御情報を管理する機能である。
- 制御情報で扱うデータを以下に記す。
  - リクエスト名
    コントローラを呼び出すための識別子。
    詳細は、『RA-01 リクエスト・コントローラマッピング機能』を参照のこと。
  - 任意の制御データ
    ユーザ名など、システムで共通に使用する情報。
    使用する制御データは、システムの要件によって異なる。

◆ 概念図



◆ 解説

(1) クライアントからリクエストをWebAPサーバに送信する。
(2) フィルタはリクエストを引数に、制御情報生成処理を実行する。
(3) 制御情報管理クラスは、リクエストに格納された情報から、リクエストに紐づく制御情報を生成し、制御情報保持クラスに登録する。
- 制御情報の管理単位
  制御情報はスレッド単位で管理される。
  １つのスレッドで１つのリクエストを処理することを想定しており、リクエストごとに一意の制御情報を扱うことができる。
  しかし、EJBを利用した場合、EJBから呼ばれるビジネスロジックは別スレッドになるため注意が必要である。

(4) フィルタはサーブレット（ディスパッチサーブレット）を実行する。
(5) フィルタは制御情報破棄処理を実行する。
(6) リクエスト終了時に、制御情報管理クラスは、リクエストに紐づく制御情報を破棄する。

■ 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のコーディングポイント
  - ➢ 制御情報管理クラスの Bean 定義

    制御情報管理クラス（RequestContextSupport 実装クラス）を定義し、制御情報を扱うクラスに設定する。

    TERASOLUNA Server Framework for Java (Rich 版)や、サービス層のクラスは、制御情報管理クラスを通して制御情報にアクセスすることができる。
    - ✧ Bean 定義サンプル

      制御情報管理クラスをコントローラに設定するサンプルを以下に記す。

      制御情報管理クラスとして、TERASOLUNA Server Framework for Java (Rich 版)が提供する DefaultRequestContextSupportImpl を使用する。

      コントローラの抽象定義に関しての詳細は、『RB-01 リクエストデータ解析機能』『RB-02 レスポンスデータ生成機能』を参照のこと。

```
<!-- 制御情報管理クラスのBean定義サンプル -->
<bean id="ctxSupport"
    class="jp.terasoluna.fw.web.rich.context.support.
    DefaultRequestContextSupportImpl"/>


<!--
  コントローラの抽象Bean定義サンプル
（受信リクエスト：XML形式、ビジネスロジック：POJO）
-->
<bean id="xmlRequestController" abstract="true">
  <property name="ctxSupport" ref="ctxSupport"/>
  <property name="dataBinderCreator" ref="xmlDataBinderCreator"/>
</bean>
```

制御情報管理クラスの定義。

制御情報管理クラスは、必ずコントローラに設定する必要がある。

- 制御情報の生成・破棄を行うフィルタの定義
  TERASOLUNA Server Framework for Java (Rich 版)が提供する制御情報の生成・破棄を行うフィルタ（RequestContextHandlingFilter）を、デプロイメントディスクリプタ（web.xml）に定義する。
  - ✧ デプロイメントディスクリプタサンプル

```
<web-app>
  (略)
  <filter>
    <filter-name>requestContextHandlingFilter</filter-name>
      <filter-class>
        jp.terasoluna.framework.web.rich.RequestContextHandlingFilter
      </filter-class>
    </filter>
    <filter-mapping>
      <filter-name>requestContextHandlingFilter</filter-name>
      <url-pattern>/*</url-pattern>
    </filter-mapping>
  (略)
```

同名で指定する。

制御情報の生成・破棄を行うフィルタの定義。

■ リファレンス

◆ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.rich.RequestContextHandlingFilter	制御情報の生成・破棄を行うサーブレットフィルタ。
2	jp.terasoluna.fw.web.rich.context.ContextSupport	リクエスト名・業務プロパティを保持するためのクラス。
3	jp.terasoluna.fw.web.rich.context. RequestContextManager	制御情報の管理クラス。 制御情報をスレッド単位で管理する。
4	jp.terasoluna.fw.web.rich.context.support. RequestContextSupport	制御情報を扱うクラスが実装すべきインタフェース。 TERASOLUNA Server Framework for Java (Rich 版)やサービス層のクラスは、本インタフェースを呼び出して、制御情報を参照することが出来る。
5	jp.terasoluna.fw.web.rich.context.support. AbstractRequestContextSupport	RequestContextSupport を継承した抽象クラス。
6	jp.terasoluna.fw.web.rich.context.support. DefaultRequestContextSupportImpl	制御情報を扱うための補助ロジックのデフォルト実装クラス。
7	org.springframework.web.servlet.DispatcherServlet	Spring が提供する、一連の処理フローを実行するサーブレット。

◆ 拡張ポイント

TERASOLUNA Server Framework for Java (Rich 版)が提供するデフォルトRequestContextSupport 実装クラス（DefaultRequestContextSupportImpl）は、リクエストに格納されたリクエスト名を制御情報に格納する処理のみを行なう。
システムの要件に応じて、業務で使用するプロパティを制御情報に設定する場合、DefaultRequestContextSupportImpl クラスを継承して、doGenerateContext()メソッドをオーバーライド実装する。
下記は、ベーシック認証等によって、認証されたユーザ名と権限を制御情報に格納する実装例である。

➢ DefaultRequestContextSupportImpl 継承クラスサンプル
フィルタ実行時に、認証されたユーザ名と権限をリクエストから取得して、制御情報に格納する。

```
public class SampleRequestContextSupportImpl extends
DefaultRequestContextSupportImpl {
  /**
   * 認証されたユーザ名と権限を、制御情報に設定する。
   *
   * @param request HTTPリクエスト
   * @return 生成した空の制御情報。
   */
  @Override
  protected RequestContext doGenerateContext( HttpServletRequest request) {
    RequestContext ctx = super.doGenerateContext(request);

    // 認証されたユーザ名を制御情報に設定する。
    ctx.setPropertyString(
        Constants.USERNAME_KEY, request.getUserPrincipal().getName());

    // 認証されたユーザの権限を制御情報に設定する。
    if(request.isUserInRole("manager")){
      ctx.setPropertyString(Constants.USERROLE_KEY, "manager");
    }
    else{
      ctx.setPropertyString(Constants.USERROLE_KEY, "all");
    }
      return ctx;
  }
}
```

DefaultRequestContextSupportImpl を継承する。

認証されたユーザ名をリクエストから取得して、制御情報に格納する。

認証されたユーザの権限をリクエストから取得して、制御情報に格納する。

➢ 定数クラスサンプル

```
public class Constants{
  /**
   * 制御情報から、認証されたユーザ名を取得するためのキー
   */
  public static final String USERNAME_KEY = "username";

  /**
   * 制御情報から、認証されたユーザの権限を取得するためのキー
   */
  public static final String USERROLE_KEY = "userrole";

}
```

定数クラス。

➢ 制御情報を扱うビジネスロジックサンプル
制御情報管理クラスを設定するための ctxSupport 属性と、その setter/getter を定義する。
属性に設定された制御情報管理クラスを用いて、認証されたユーザ名と権限を制御情報から取得する
※DI コンテナの機能を用い、制御情報管理クラスを属性に設定する。制御情報管理クラスを属性に設定する方法は、下記 Bean 定義サンプルを参照のこと。

```
public class SumServiceImpl implements SumService {

  /**
   * 制御情報管理クラス
   */
  protected RequestContextSupport ctxSupport = null;

  /**
   * 制御情報管理クラスのsetter
   */
  public void setCtxSupport(RequestContextSupport ctxSupport) {
    this.ctxSupport = ctxSupport;
  }

  /**
   * 制御情報管理クラスのgetter
   */
  public RequestContextSupport getCtxSupport() {
    return ctxSupport;
  }

  /**
  * 業務ロジック
  */
  public SumResult sum(SumParam sumParam) throws Exception {
    SumResult result = new SumResult();
    // 認証されたユーザ名を取得する
    String userName = ctxSupport.getPropertyString(Constants.USERNAME_KEY);
    // 認証されたユーザの権限を取得する
    String userRole = ctxSupport.getPropertyString(Constants.USERROLE_KEY);
    （略）
    return result;
  }
}
```

制御情報管理クラスが設定される属性。

属性に設定された制御情報管理クラスを用いて、
認証されたユーザ名と権限を制御情報から取得する。

- Bean 定義サンプル
  上記で生成した制御情報管理クラス（SampleRequestContextSupportImpl）、ビジネスロジック（SumServiceImpl）、ビジネスロジックを呼び出すコントローラの Bean 定義を行う。
  コントローラの Bean 定義の詳細は、『RB-01 リクエストデータ解析機能』『RB-02 レスポンスデータ生成機能』を参照のこと。

```
<!-- 制御情報管理クラス（SampleRequestContextSupportImpl）のBean定義サンプル -->
<bean id="ctxSupport"
  class="jp.terasoluna.sample2.context.SampleRequestContextSupportImpl"/>

<!-- コントローラのBean定義サンプル -->
<bean name="/sumController"
    class="jp.terasoluna.sample2.web.controller.SumController"
    parent="pojoXmlRequestCastorViewController" scope="prototype">
  <property name="sumService" ref="sumService"/>
</bean>

<!-- 制御情報を扱うビジネスロジック（SumServiceImpl）のBean定義サンプル -->
<bean name="sumService"
    class="jp.terasoluna.sample2.service.impl.SumServiceImpl"
    scope="prototype">
  <property name="ctxSupport" ref="ctxSupport"/>
</bean>
```

ビジネスロジックに、制御情報管理クラスを設定する。

■ 関連機能

- 『RA-01 リクエスト・コントローラマッピング機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Rich 版) 機能網羅サンプル
  - UC104 制御情報管理
    - jp.terasoluna.rich.functionsample.informationcontrol.*

- TERASOLUNA Server Framework for Java (Rich 版) チュートリアル
  - /webapps/WEB-INF/applicationContext.xml 等

■ 備考

- なし

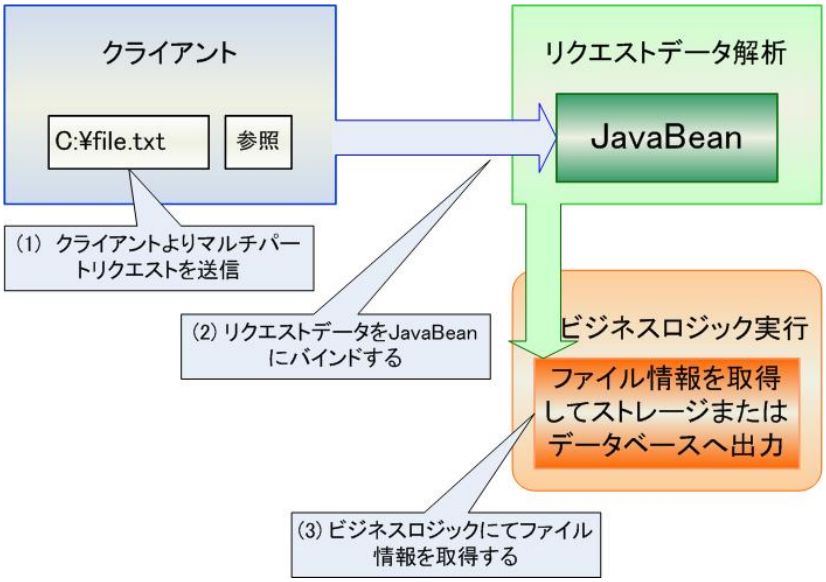
RB-05 ファイルアップロード機能

■ 概要

◆ 機能概要

- クライアントからファイルをアップロードする機能を提供する。
  - Spring MVC Framework の multipart support 機能をほぼそのまま使用する。

◆ 概念図



◆ 解説

(1) クライアントは、マルチパートリクエストを送信する。
(2) リクエスト解析機能は、マルチパートリクエストに含まれるアップロードファイルの情報をJavaBeanにバインドする。
(3) ビジネスロジック実行機能は、アップロードファイルの情報を含んだJavaBeanを引き数にして、業務ロジックを実行する。業務ロジックは、業務要件に応じて、ストレージへの保存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

- クライアントの実装
  - マルチパートリクエストの送信処理を実装する。

    TERASOLUNA Server Framework for Java (Rich版)は、RFC1867(*Form-based File Upload in HTML :* http://www.ietf.org/rfc/rfc1867.txt)に準拠した、HTMLのフォームを利用してファイルデータをPOST する形式のHTTPリクエストを解析することができる。クライアントは、RFC1867 に従う形式でリクエスト送信処理を実装しなければならない。

    具体的なリクエストの送信処理の実装方法は、適用するクライアントアプリケーションによって異なるので、各プロダクトのドキュメントを参照すること。

    以下はHTMLフォームを利用してファイルアップロードを行う場合のHTMLテキストの記述例である。

```
・・・
<body>
   <form action="http://localhost:8080/sample/secure/blogic.do"
     method="POST" enctype="multipart/form-data">
       <input type="file" name="file1"/>
       <input type="file" name="file2"/>
       <input type="text" name="text1"/>
       <input type="submit"/>
   </form>
</body>
・・・
```

    - ✧ リクエストヘッダ Content-Type の指定時の注意点

      マルチパートリクエストを送信する場合は、ヘッダの Content-Type に以下のような文字列を設定し、multipart/form-data のメディアタイプで送信する必要がある。

      以下の”BOUNDARY”という文字列は、マルチパートリクエストの各パートの区切りとなる文字列である。ファイルデータ中に現れない任意の文字列を指定する必要がある。

```
multipart/form-data;boundary=BOUNDARY
```

- ✧ その他のリクエストヘッダ設定時の注意点
  TERASOLUNA Server Framework for Java (Rich 版)のリクエストデータ解析以外の機能は、アップロード／非アップロードを意識せず統一的に処理を行うことができる。
  したがってクライアントは、Content-Type の他にも、通常の非ファイルアップロード時のリクエストヘッダに設定する制御情報を、アップロード時もヘッダに設定する必要がある。
  たとえば、通常、リクエスト名をリクエストヘッダに設定しているシステムでは、ファイルアップロード時にも、マルチパートリクエストのヘッダにリクエスト名を設定しなければならない。

- ✧ ファイルパラメータ設定時の注意点
  日本語ファイル名のファイルをアップロードする場合には、あらかじめ、システムごとにサーバ・クライアント間でのエンコーディングの取り決めをしておく必要がある。サーバがリクエストのファイル名を読み込む際に利用する文字コードの設定と、クライアントがファイル名文字列をリクエストのストリームに書き込む際に利用する文字コードを一致させるなどの工夫で、ファイル名の文字化けの発生を防ぐことができる。詳細は、後述の「マルチパート解決クラスの利用」を参考にすること。

- マルチパート解決クラスの利用
  - ➢ Bean定義ファイルにマルチパート解決クラスのBean定義を行う。
    マルチパート解決クラスは、マルチパートリクエストの解析を行う責務を持つ。以下の設定では、ファイルアップロード用のオープンソースパッケージ Commons FileUpload (http://jakarta.apache.org/commons/fileupload)に実処理を委譲している。
    - ✧ TERASOLUNA Server Framework for Java (Rich 版)を利用したアプリケーション開発時の雛形となるブランクプロジェクトでは、Bean 定義ファイルに以下の設定がコメントアウトされて記述されている。コメントアウトを解除して定義を有効にする。
    - ✧ また、日本語ファイルのアップロードを行う場合には、defaultEncoding プロパティの指定を行う。
      以下は、クライアントがファイル名文字列をバイト化してリクエストのストリームに書き込むのに”utf-8”の文字セットを利用している場合、クライアントにあわせて defaultEncoding プロパティに”utf-8”を指定する例である。

```
<!-- クエリ形式データバインダ用マルチパート解決クラス -->
<bean id="multipartResolver"
  class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
  <!-- 最大ファイルサイズ(単位：バイト) -->
  <property name="maxUploadSize" value="100000"/>
  <property name="defaultEncoding" value="utf-8" />
</bean>
```

- JavaBean の作成
  - マルチパートリクエストの情報をバインドする先の JavaBean を作成する。
    - ファイルパラメータを格納する属性を定義する。
      クライアントで指定したファイル形式のフィールド名(HTML から送信した場合、type 属性が”file”である<input>タグの name 属性の値にあたる部分)と同名の属性を byte 配列型または MultipartFile 型で作成する。
      リクエストデータ解析機能にて、作成した属性には、DataBinder にて、アップロードされたファイルの情報が格納される。
    - テキストパラメータを格納する属性を定義する。
      クライアントからテキスト形式の項目もリクエストしていた場合は、フィールド名(HTML から送信した場合、type 要素が”text”である<input>タグの name 属性の値にあたる部分)と同名の属性を String で作成する。
      リクエストデータ解析機能にて、作成した属性には、DataBinder にて、テキストの情報が格納される。

```
import org.springframework.web.multipart.MultipartFile;

public class FileUploadInput {
    private byte[] file1 = null;
    private MultipartFile file2 = null;
    private String text1 = null;

    public void setFile1(byte[] file1) {
        this.file1 = file1;
    }
    public byte[] getFile1() {
        return file1;
    }
    public void setFile2(MultipartFile file2) {
        this.file2 = file2;
    }
    public MultipartFile getFile2() {
        return file2;
    }
    public void setText1(String text1) {
        this.text1 = text1;
    }
    public String getText1() {
        return text1;
    }
}
```

✧ 上記に示したとおり、ファイル情報を格納する属性は、byte 配列型または MultiltipartFile 型で宣言することが可能である。どちらの型を採用するかについては、以下の比較を参考にして、適用プロジェクトにて判断する必要がある。

	メリット	デメリット
byte 配列型	○JavaBean が Spring の API に依存しないため、Spring を利用しないシステムでの JavaBean の再利用が可能	×巨大なデータを扱うことができない。 ×ファイルデータ以外の情報を取得できない。
MultiltipartFile 型	○ファイルデータをストリーム形式で取得できるため、巨大なデータを扱うことが可能 ○ファイルデータ以外にも、ファイルのコンテントタイプ、オリジナルファイル名などの取得が可能。	×JavaBean が Spring の API に依存するため、Spring を利用しないシステムでの JavaBean の再利用が不可能

- コントローラの作成
  - コントローラの Bean 定義を行う。
    XML 形式ではなく、クエリ形式のリクエストを受信するためのコントローラ抽象定義の bean 名称を parent 属性に指定しなければならない。
    その他の設定は、非アップロード時の Bean 定義と同様である。詳細は、『WA01 コントローラ拡張機能』および『WC01 ビジネスロジック実行機能』を参照のこと。

```
<bean name="/fileUploadController"
    class="FileUploadController"
    parent="pojoQueryRequestCastorViewController"
    scope="prototype">
  <property name="fileUploadService" ref="fileUploadService"/>
</bean>
```

- コントローラを実装する。
  コントローラの実装方法は、非アップロード時と同様である。詳細は、『WA01 コントローラ拡張機能』および『WC01 ビジネスロジック実行機能』を参照のこと。

```
public class FileUploadController extends
TerasolunaController<FileUploadInput, FileUploadOutput> {
    FileUploadService service = null;
    . . .
    @Override
    protected FileUploadOutput executeService(FileUploadInput input)
        throws Exception {
        return service.execute();
    }
}
```

- ビジネスロジックの作成
  - 業務ロジックでは、引き数に設定されたファイルの情報を使用して、要件に応じてストレージへの保存処理やデータベースへの登録処理を行う。
    以下に業務ロジックの実装例を示す

```
import java.io.IOException;
import java.io.InputStream;

import org.springframework.web.multipart.MultipartFile;

public class FileUploadService {
   public FileUploadOutput execute(FileUploadInput input) {
      // バイト配列型の属性からファイルデータ取得
      byte[] file1 = input.getFile1();

      // MultipartFile型の属性からファイルデータ取得
      MultipartFile file2 = input.getFile2();
      // ファイルデータをバイト配列で取得(巨大ファイル取扱時には利用しないこと)
      try {
         byte[] file2bytes = file2.getBytes();
      } catch (IOException e) {
         // 省略
      }

      // ファイルデータをストリーム形式で取得
      InputStream is = null;
      try {
         is = file2.getInputStream();
      } catch (IOException e) {
         // 省略
      } finally {
         if (is == null) {
            try {
               is.close();
            } catch (IOException e) {
               // 省略
            }
         }
      }

      // ファイルのコンテントタイプを取得
      String contentType = file2.getContentType();
      // クライアントで指定されたファイル名を取得
      String fileName = file2.getOriginalFilename();
      // ファイルサイズを取得
      long size = file2.getSize();

      return new FileUploadOutput();
   }
}
```

■ 拡張ポイント

なし。

■ 関連機能

- 『RA-02 コントローラ拡張機能』
- 『RB-01 リクエストデータ解析機能』
- 『RC-01 ビジネスロジック実行機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Rich 版) 機能網羅サンプル
  - UC-113 ファイルアップロードコントローラ
    - jp.terasoluna.rich.functionsample.upload.*

■ 備考

なし。

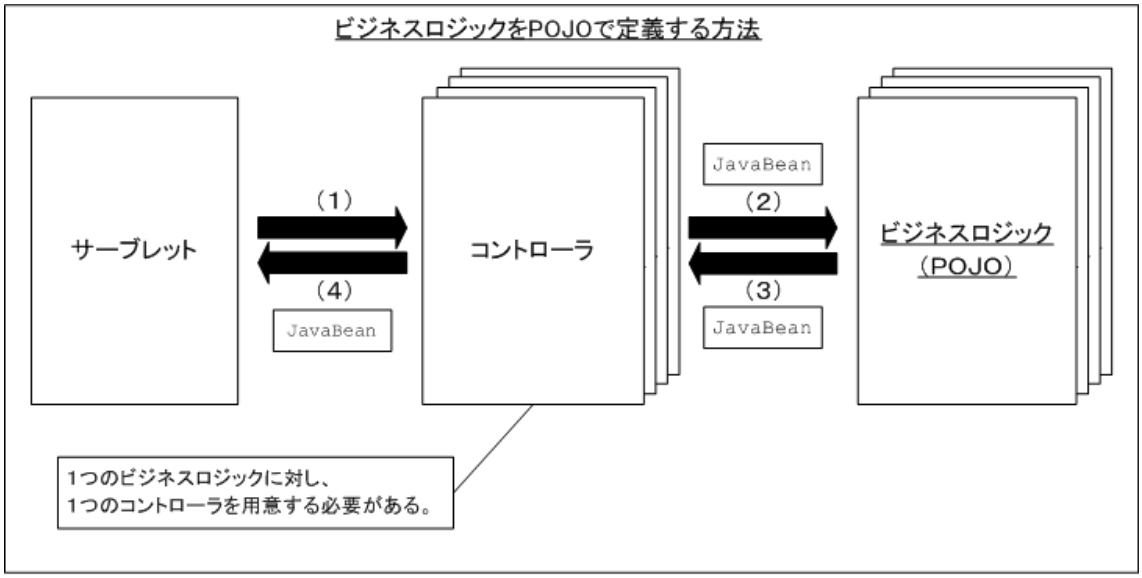
RC-01 ビジネスロジック実行機能

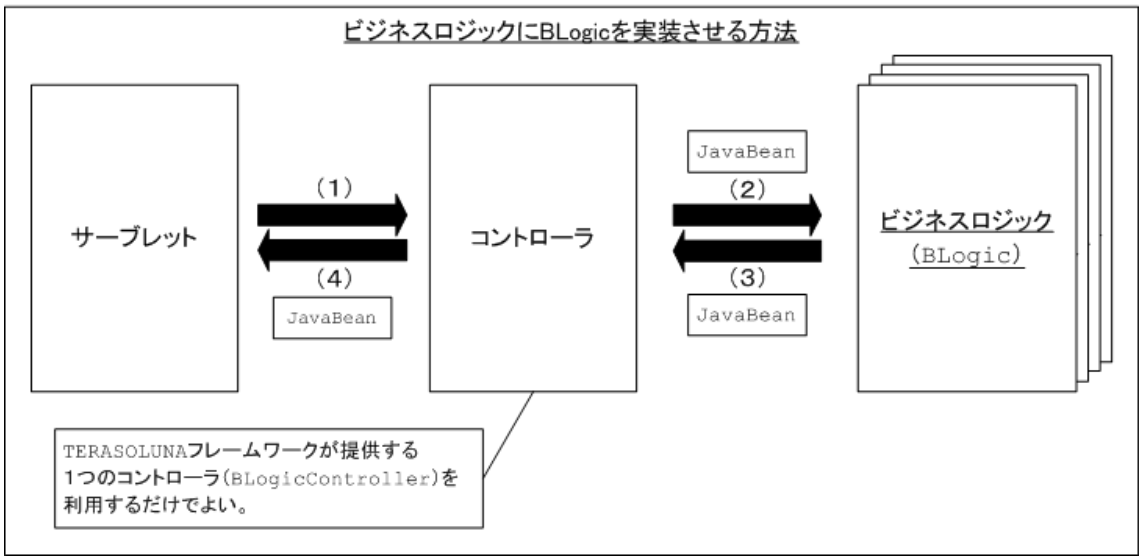
■ 概要

◆ 機能概要

- コントローラがビジネスロジック（サービス層のクラス）を呼び出す機能である。
- TERASOLUNA Server Framework for Java (Rich 版)では以下の２種類の方法をサポートしており、適宜選択すること。
  - POJO で定義されたビジネスロジックを呼び出す方法。
  - TERASOLUNA Server Framework for Java (Rich 版)が提供するビジネスロジック用インタフェースを実装したビジネスロジックを呼び出す方法。

◆ 概念図





◆ 解説

(1) サーブレット（ディスパッチャサーブレット）はコントローラを実行する。
(2) コントローラは、JavaBeanを引数にビジネスロジックを実行する。
ビジネスロジックの実現方法として、以下の２種類が挙げられる。

- ビジネスロジックを POJO で定義する方法
  TERASOLUNA Server Framework for Java (Rich 版)が提供する抽象コントローラ（TerasolunaController）の実装クラスから、POJO を実行する。
  TerasolunaController に関しての詳細は『RA-02 コントローラ拡張機能』を参

照のこと。
この方法を使用した場合の、メリット・デメリットを以下に記す。
- ✧ メリット
  - ● ビジネスロジックを POJO で実装することができる。
  - ● 自由なインタフェース設計が可能。
- ✧ デメリット
  - ● ビジネスロジックの数だけコントローラを作成する必要がある。

➢ ビジネスロジックに TERASOLUNA Server Framework for Java (Rich 版)提供のビジネスロジック用インタフェース（BLogic）を実装する方法
TERASOLUNA Server Framework for Java (Rich 版) が提供する TerasolunaController 実装クラス（BLogicController）から、BLogic を実行する。
この方法を使用した場合のメリット・デメリットを以下に記す。
- ✧ メリット
  - ● 各ビジネスロジックを実行するコントローラが 1 つで良い。
- ✧ デメリット
  - ● ビジネスロジックが TERASOLUNA Server Framework for Java (Rich 版)に依存するため、再利用性が低下する。

(3) ビジネスロジックは、実行結果をJavaBeanとして返却する。
(4) コントローラは、(3)で返却されたJavaBeanを返却する。

■ 使用方法

◆ コーディングポイント

- POJO で定義されたビジネスロジックのコーディングポイント
  - ビジネスロジックを POJO で定義した場合
    - ✧ コントローラの抽象定義
      POJO で定義されたビジネスロジックを実行するコントローラの抽象 Bean 定義を有効化する。詳細は『RA-02 コントローラ拡張機能』を参照のこと。

- 業務開発者のコーディングポイント
  - ビジネスロジックが実装すべきインタフェースの生成
    - インタフェースサンプル

```
public interface SumService {
  SumResult sum(SumParam sumParam) throws Exception;
}
```

コントローラから実行されるメソッド。

  - ビジネスロジック（POJO）の生成

上記インタフェースを実装した、ビジネススロジックを生成する。

    - ビジネスロジックサンプル

```
public class SumServiceImpl implements SumService {
  public SumResult sum(SumParam sumParam) throws Exception {
    SumResult result = new SumResult();
    result.setResult((sumParam.getParam1() + sumParam.getParam2()));
    return result;
  }
}
```

戻り値の型。

引数の型。

  - ビジネスロジックを実行するコントローラの生成

TERASOLUNA Server Framework for Java (Rich 版)が提供する抽象コントローラ（TerasolunaController）の実装クラスを生成する。
ビジネスロジックの引数及び戻り値となる JavaBean の型を定義する。
※JavaBean の型として、java.lang.Object やインタフェースの型を設定することはできない（もし設定した場合、例外がスローされる）。なお、JavaBean には引数のないコンストラクタと、属性のアクセサメソッド（getter,getter）を用意すること。

    - コントローラサンプル

```
public class SumController
  extends TerasolunaController<SumParam, SumResult> {

  private SumService sumService;

  public void setSumService(SumService sumService) {
    this.sumService = sumService;
  }

  @Override
  protected SumResult executeService(SumParam ivo) {
    return sumService.sum(ivo);
  }
}
```

TerasolunaController を継承する。

ビジネスロジックへ渡す引数の型。

ビジネスロジックから返却される戻り値の型。

実行するビジネスロジックのインタフェース。

executeService メソッドを実装し、ビジネスロジックを実行する。

✧ ビジネスロジック・コントローラの Bean 定義

ビジネスロジック（POJO）と、それを呼び出すコントローラ（TerasolunaController 実装クラス）の Bean 定義を行う。

● Bean 定義サンプル

```
<!--
コントローラ（TerasolunaController実装クラス）のBean定義サンプル
-->
<bean name="/sumController"
    class="jp.terasoluna.sample2.web.controller.SumController"
    parent="pojoXmlRequestCastorViewController" scope="prototype">
  <property name="sumService" ref="sumService"/>
</bean>

<!-- ビジネスロジック（POJO）のBean定義サンプル-->
<bean id="sumService“
  class="jp.terasoluna.sample2.service.impl.SumServiceImpl"
  scope="prototype">
</bean>
```

ソフトウェアアーキテクトが抽象 Bean 定義したコントローラを設定する。

コントローラにビジネスロジック（POJO）を設定する。

スレッドセーフな設計が行われている場合は、scope 属性を singleton に設定する。

- ビジネスロジックに TERASOLUNA Server Framework for Java (Rich 版)提供のビジネスロジック用インタフェース（BLogic）を実装した場合
  - ソフトウェアアーキテクトのコーディングポイント
    BLogicController の抽象 Bean 定義を有効化する。詳細は『RA-02 コントローラ拡張機能』を参照のこと。

- 業務開発者のコーディングポイント
  - ビジネスロジックの生成

    BLogic インタフェースを実装したビジネスロジックを生成する。
    ビジネスロジックの引数及び戻り値となる JavaBean の型を定義する。
    ※JavaBean の型として、java.lang.Object やインタフェースの型を設定することはできない（もし設定した場合、例外がスローされる）。なお、JavaBean には引数のないコンストラクタと、属性のアクセサメソッド（getter,setter）を用意すること。

    - ビジネスロジックサンプル

```
public class SumBLogic implements BLogic<SumParam, SumResult> {

public SumResult execute(SumParam sumParam) {
  SumResult result = new SumResult();
  result.setResult((sumParam.getParam1() + sumParam.getParam2()));
  return result;
}
```

引数の型。

戻り値の型。

BLogic インタフェースを実装する。

  - ビジネスロジック・コントローラの Bean 定義

    ビジネスロジック（BLogic）と、それを呼び出す BLogicController の Bean 定義を行う。

    - Bean 定義サンプル

```
<!-- コントローラ（BLogicController）のBean定義-
<bean name="/sumController"
    parent="xmlRequestBLogicExecuteController" scope="prototype">
  <property name="blogic" ref="sumBLogic"/>
</bean>

<!-- ビジネスロジック（BLogic）のBean定義サンプル -->
<bean id="sumBLogic"
    class="jp.terasoluna.sample2.service.blogic.SumBLogic"
    scope="prototype"/>
</bean>
```

ソフトウェアアーキテクトが抽象 Bean 定義した、コントローラを設定する。

コントローラにビジネスロジック（BLogic）を設定する。

スレッドセーフな設計が行われている場合は、scope 属性を singleton に設定する。

■ リファレンス

◆ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.rich.springmvc.controller.TerasolunaController	サービス層のクラスを実行するコントローラ抽象クラス。
2	jp.terasoluna.fw.web.rich.springmvc.controller.BLogicController	BLogic インタフェース実装クラス実行用コントローラ。
3	jp.terasoluna.fw.service.rich.BLogic	BLogicController から実行されるビジネスロジックが実装すべきインタフェース。
4	org.springframework.web.servlet.DispatcherServlet	Spring が提供する、一連の処理フローを実行するサーブレット。コントローラを実行する。

◆ 拡張ポイント

なし

■ 関連機能

- 『RA-02 コントローラ拡張機能』
- 『CA-01 トランザクション管理機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Rich 版) 機能網羅サンプル
  - UC105 サービスクラス実行

  jp.terasoluna.rich.functionsample.service.*

- TERASOLUNA Server Framework for Java (Rich 版) チュートリアル
  - 「2.3 単純なロジック」
  - /webapps/WEB-INF/tutorial-servlet.xml
  - /webapps/WEB-INF/tutorial-controller.xml
  - jp.terasoluna.rich.tutorial.service.blogic.SimpleBLogic.java 等

■ 備考

- なし

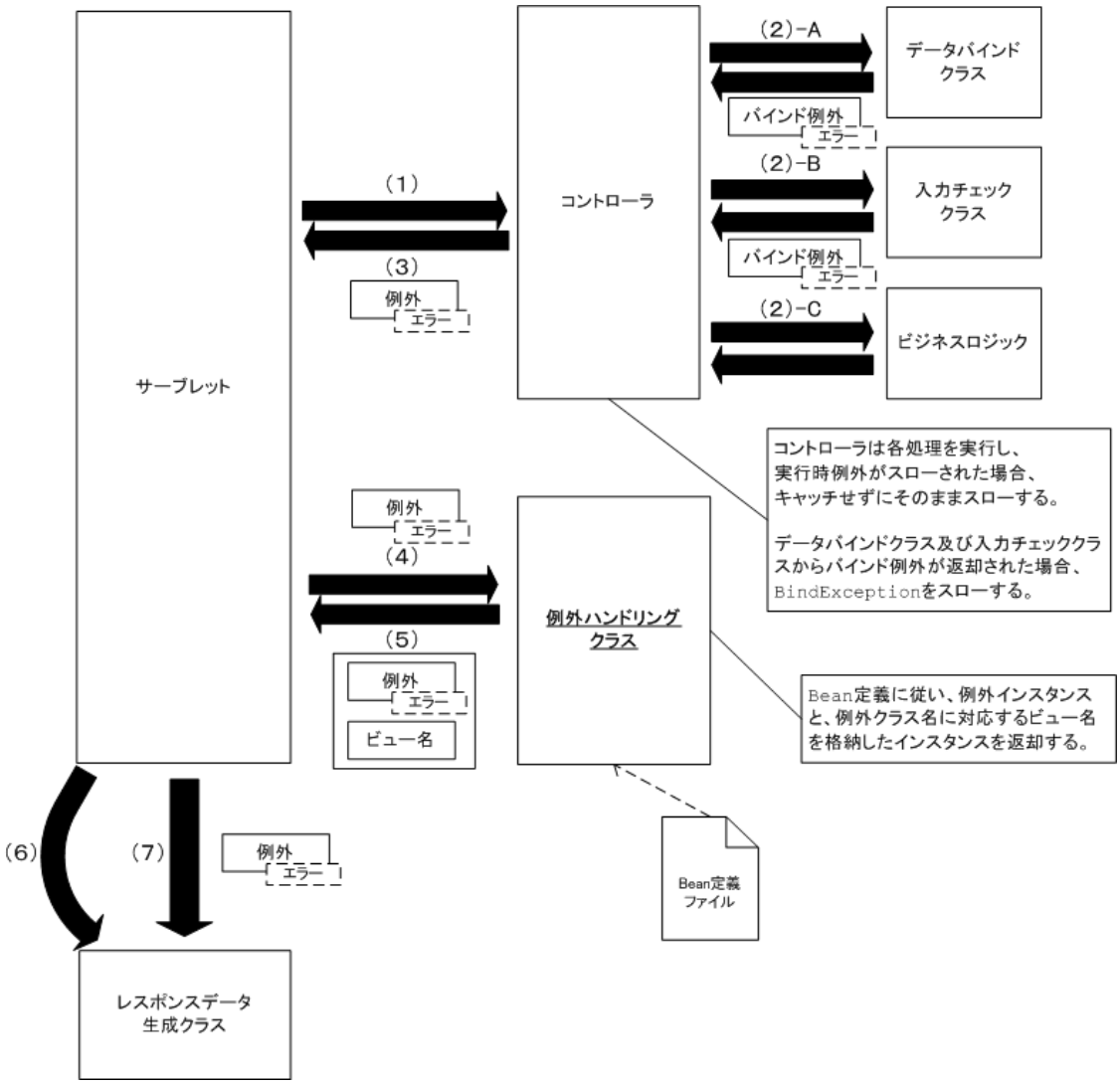
RD-01 例外ハンドリング機能

■ 概要

◆ 機能概要

- TERASOLUNA Server Framework for Java (Rich 版)において、コントローラで発生した例外をもとに、レスポンスデータを生成する機能である。
  - テンプレートファイルをもとに、エラー用のレスポンスデータを生成する
- 例外クラスごとにエラー種別とエラーコードの設定が可能である。
  - エラー種別はレスポンスヘッダに設定される。

◆ 概念図



◆ 解説

(1) サーブレット（ディスパッチャサーブレット）はコントローラを実行する。

(2)－A　コントローラはデータバインドクラスを実行する。
データバインドクラスは、形式チェック時にエラーが発生した場合、コントローラから渡されたバインド例外（BindException）にエラー情報を格納する。
データバインドクラスの詳細は、『RB-01 リクエストデータ解析機能』を参照のこと。

(2)－B　コントローラは入力チェッククラスを実行する。
入力チェッククラスは、入力チェック時にエラーが発生した場合、コントロ

ーラから渡されたバインド例外（BindException）にエラー情報を格納する。
コントローラの詳細は、『RA-02 コントローラ拡張機能』を参照のこと。

(2)－C コントローラはビジネスロジックを実行する。
ビジネスロジックの詳細は、『RC-01 ビジネスロジック実行機能』を参照のこと。

(3) コントローラは以下の場合、サーブレットに例外をスローする。
- (2)で実行時例外がスローされた場合
  実行時例外はキャッチせず、そのままスローする。
- (2)でバインド例外にエラーが格納されていた場合
  BindExceptionをスローする。
  ※BindExceptionとは、Springが提供している例外クラスである。データバインド及び入力チェック時に発生したエラー情報を格納するために使用する。

(4) サーブレットは、コントローラでスローされた例外をキャッチした場合、例外を引数として、例外ハンドリングクラスを実行する。

(5) 例外ハンドリングクラスはBean定義に従い、引数で渡された例外とそのビュー名及びエラーコードを返却し、レスポンスヘッダにエラー種別を設定する。

(6) サーブレットは、(5)で返却されたビュー名をもとに、ビュー（レスポンスデータ生成クラス）を生成する。
詳細は、『RB-02 レスポンスデータ生成機能』を参照のこと。

(7) サーブレットは、(6)で生成したビューを実行する。
詳細は、『RB-02 レスポンスデータ生成機能』を参照のこと。

■ 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
  - 例外ハンドリングクラスの Bean 定義
    例外に対応するビュー名、エラー種別、エラーコードを定義する。
    ※エラーコードは必ずしも定義する必要はなく、ビューで使用する場合に設定を行う。
    - 例外の対応付けルール
      Bean 定義を行った順番（上から順）に例外の評価が行われる。
      - 下記の Bean 定義ファイルの設定例の場合、設定された順番（ OXMappingException → UnknownRequestNameException → SystemException→BindException→Exception）に沿って評価が行われ、合致した時点でビュー名が決定される。
        例えば、コントローラ実行時に OXMappingException がスローされた場合、“oxmException”というビュー名と、“8004C028”というエラーコードがサーブレット（ディスパッチャサーブレット）に返却され、レスポンスヘッダにエラー種別“exception”が設定される。

✧ Bean 定義サンプル

```
<!-- 例外ハンドリングクラスのBean定義サンプル -->
<bean id="handlerExceptionResolver"
    class="jp.terasoluna.fw.web.rich.springmvc.servlet.handler.
    SimpleMappingExceptionResolverEx">
  <property name="linkedExceptionMappings">
    <map>
      <entry key="jp.terasoluna.fw.oxm.exception.OXMappingException">
        <value>oxmException,exception, errors.8004C028</value>
      </entry>
      <entry key="jp.terasoluna.fw.web.rich.exception.
          UnknownRequestNameException">
        <value>exception,exception, errors.800
      </entry>
      <entry key="jp.terasoluna.fw.exception.S
        <value>systemException,exception</valu
      </entry>
      <entry key="org.springframework.validation.BindException">
        <value>bindException,validation</value>
      </entry>
      <entry key="java.lang.Exception">
        <value>exception,exception, errors.8004C999</va
      </entry>
    </map>
  </property>
</bean>
```

例外とビュー名の対応付けを行う HandlerExceptionResolver 実装クラス。

ハンドリングされる例外クラス名。

エラー種別。

エラーコード。

発生した例外に対して使用されるビュー名。
ビュー名に対応する Velocity テンプレートファイルを生成する必要がある。
詳細は、後述の「例外に対応する Velocity テンプレートファイルの生成」を参照のこと。

バインド例外のハンドリング。
詳細は、後述の「テンプレートファイルサンプル２」を参照すること。

全ての例外は java.lang.Exception を継承しているので、明示的に定義されていない例外が発生した場合でも、必ず最後の java.lang.Exception の設定で例外ハンドリングが行われる。

➢ 例外に対応する Velocity テンプレートファイルの生成

TERASOLUNA Server Framework for Java (Rich 版)では、例外のレスポンスデータ生成に Velocity ビューを使用する。

Bean 定義したビュー名に対応する Velocity テンプレートファイルを生成し、指定されたパスに配置する。

ファイル名は「ビュー名＋拡張子（.vm)」の式で求められる。

例えば、ビュー名が“exception”の場合、ファイル名は「exception.vm」となる。

テンプレートファイルの生成方法は、『RB-02 レスポンスデータ生成機能』を参照のこと。

✧ バインド例外からのエラー情報の取得

バインド例外（BindException）には、形式チェッククラスおよび入力チェッククラスから返却されたエラー情報の配列が格納されており、それらを順番に取り出して置換する必要がある。

エラー情報の配列は変数 exception を通して取得でき、Velocity の機能を用いて、エラー情報を順番に取得して表示する。

● エラー情報の内容を以下に記す。

変数名	概要
arguments	置換文字列の配列。
field	エラーが発生したフィールド情報。
code	エラーコード。

✧ テンプレートファイルサンプル 1

上記 Bean 定義のビュー名“exception”に対応するテンプレートファイルの例を以下に記す。

➢ exception.vm

```
<?xml version="1.0" encoding="utf-8" ?>
<errors>
    <error>
##   エラーコード
    <error-code>$!{errorCode}</error-code>
##   エラーコードに対応するメッセージ
    <error-message>
      $!{rc.getMessage("${errorCode}")}
    </error-message>
  </error>
</errors>
```

Bean 定義ファイルに設定したエラーコードに置換される。

エラー情報を引数に、リクエストコンテキストの getMessage メソッドを実行することで、リソースバンドルを利用したメッセージの解決を行なうことができる。Velocity ビューでリクエストコンテキストを使用する方法は、『RB-02 レスポンスデータ生成機能』を参照のこと。

- メッセージリソースサンプル（error-messages.properties.sjis）
  以下のエラーコードとメッセージが定義されていることとする。

```
errors.8004C003 = 定義されていないリクエストです。
```

- メッセージリーソースの設定。
  プロパティファイルからメッセージを取得する。（複数ファイル可）
  以下に messageSource の Bean 定義設定例を記す。

```
<!-- メッセージの設定 -->
<bean id="messageSource“
   class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames“ value=“applicationResources,error-messages”/>
</bean>
```

プロパティファイルを指定する。

- 生成されるレスポンスデータ
  上記設定の場合、UnknownRequestNameException がスローされると、“errors.8004C003”のエラーコードと、エラーコードに対応するメッセージに置換されたレスポンスデータが生成される。

```
<?xml version=”1.0” encoding=”utf-8” ?>
<errors>
   <error>
##   エラーコード
   <error-code> errors.8004C003</error-code>
##   エラーコードに対応するメッセージ
   <error-message>定義されていないリクエストです。</error-message>
  </error>
</errors>
```

置換されたエラーコード。

置換されたメッセージ。

➢ テンプレートファイルサンプル２

上記 Bean 定義のビュー名“bindException”に対応するテンプレートファイルの例を以下に記す。

➢ bindException.vm

```
<?xml version="1.0" encoding="utf-8" ?>

<errors>
#foreach($error in $exception.allErrors)
  <error>
    #foreach($arg in $error.arguments)
##    置換文字列
      <replace-values>${arg}</replace-values>
    #end
##    エラーが発生したフィールド情報
    <error-field>${error.field}</error-field>
##    エラーコードに対応するメッセージ
    <error-message>${rc.getMessage(${error})}</error-message>
##    エラーコード
    <error-code>${error.code}</error-cod
  </error>
#end
</errors>
```

Velocity の機能を利用して変数 exception に格納されたエラー情報の配列から、エラー情報を順番に取り出して、変数 error に格納する。

Velocity の機能を利用して、変数 arguments に格納された置換文字列の配列から、置換文字列を順番に取り出して、変数 arg に格納する。

置換文字列。

エラーが発生したフィールドの情報。

エラー情報を引数に、リクエストコンテキストの getMessage メソッドを実行することで、リソースバンドルを利用したメッセージの解決を行なうことができる。Velocity ビューでリクエストコンテキストを使用する方法は、『RB-02 レスポンスデータ生成機能』を参照のこと。

エラーコード。

➢ メッセージリソースサンプル（errors.properties.sjis）

以下のエラーコードとメッセージが定義されていることとする。

```
typeMismatch.number = {0}には{1}値を入力してください.
```

エラー情報のエラーコード。

エラーコードに対応するメッセージ。
エラー情報の置換文字列でプレースホルダ（例中の{0}, {1}）が置換される。

➢ 生成されるレスポンスデータ

上記設定で、コントローラから BindException がスローされた場合、生成されるレスポンスデータを以下に記す。

✧ BindException に格納されているエラー情報

コントローラからスローされた BindException に格納されているエラー情報に、以下の値が設定されていることとする。

変数名	値
arguments	{sample-dto[0].user-id[0], integer, 00a1}
field	sample-dto[0].user-id[0]
code	typeMismatch.number

✧ 生成されるレスポンスデータ

```
<?xml version="1.0" encoding="utf-8" ?>

<errors>
 <error>
  <replace-values>sample-dto[0].user-id[0]</replace-values>
  <replace-values>integer</replace-values>
  <replace-values>00a1</replace-values>
  <error-field>sample-dto[0].user-id[0]</error-field>
  <error-message>
    sample-dto[0].user-id[0]にはinteger値を入力してください.
  </error-message>
  <error-code>typeMismatch.number</error-code>
 </error>

</errors>
```

置換された置換文字列の配列。

置換されたフィールド情報。

置換されたメッセージ。

置換されたエラーコード。

エラー情報の配列の数だけ、
infos 要素以下を繰り返し出力する。

■ リファレンス

◆ 構成クラス

	クラス名	概要
1	org.springframework.web.servlet.handler.SimpleMappingExceptionResolver	Spring が提供する、例外のハンドリングを行う HandlerExceptionResolver 実装クラス。
2	jp.terasoluna.fw.web.rich.springmvc.servlet.handler.SimpleMappingExceptionResolverEx	SimpleMappingExceptionResolver を拡張し、Bean 定義の例外設定に順序性を持たせる機能、エラーコードを扱う機能等を追加したクラス。
3	org.springframework.web.servlet.DispatcherServlet	Spring が提供する、一連の処理フローを実行するサーブレット。リクエストとコントローラのマッピング、コントローラの実行、ビューの生成を行う。
4	org.springframework.web.servlet.ModelAndView	Spring が提供する、コントローラから返却された例外とビュー名を保持するクラス。
5	org.springframework.validation.BindException	データバインドクラス及び形式チェッククラスから返却されるエラー情報を格納するクラス。
6	org.springframework.web.servlet.support.RequestContext	Spring が提供するコンテキストクラス。本クラスを使用することで、メッセージリソースや例外オブジェクトへ容易にアクセスすることが可能となる。
7	jp.terasoluna.fw.web.rich.springmvc.servlet.handler.ExceptionResolveDelegator	エラー情報をレスポンスヘッダと ModelAndView に反映するためのインタフェース。
8	jp.terasoluna.fw.web.rich.springmvc.servlet.handler.ExceptionResolveDelegatorImpl	ExceptionResolveDelegator のデフォルト実装クラス。

◆ 拡張ポイント

なし

■ 関連機能

- 『RA-02 コントローラ拡張機能』
- 『RB-01 リクエストデータ解析機能』
- 『RB-02 レスポンスデータ生成機能』
- 『RF-01 形式チェック機能』
- 『RF-02 入力チェック機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Rich 版) 機能網羅サンプル
  - UC106 例外処理
    - jp.terasoluna.rich.functionsample.exceptionhandler.*

- TERASOLUNA Server Framework for Java (Rich 版) チュートリアル
  - 「2.5.1 電文形式チェック」
  - 「2.5.2 単項目入力チェック」
  - 「2.5.3 相関入力チェック」
  - 「2.6 例外処理」
  - 「2.7 アクセス制御」
  - /webapps/WEB-INF/tutorial-servlet.xml
  - /webapps/WEB-INF/velocity/*
  - /sources/*-messages.properties 等

■ 備考

- SystemException と ServiceException について
  TERASOLUNA Server Framework for Java (Rich 版)には、汎用例外クラスとして SystemException と ServiceException がある。両者の使い分けは以下のようになる。

  - SystemException
    業務ロジック中で例外が発生した場合に利用する。
    発生した例外を元に、メッセージキー、メッセージ置換文字列などを設定し SystemException を生成する。

    - 使用例

```
// 共通処理（CP_B99_01）を呼び出し、ツアーに登録する情報を取得する
RegisterTourBean[] tourInfo = null;
try {
    tourInfo = ExcelBeanUtil.createRegisterTourBean(inputStream);
} catch (IOException e) {
    // ExcelBeanUtilクラスの引数のストリームが不正な場合
    throw new SystemException(e, "errors.8004C032");
}
```

  - ServiceException
    サービス層のクラスで、業務エラーが発生した場合に利用する。
    例えば、業務的にデータの不整合等が起きている場合に利用する。

    - 使用例

```
// 認証
Object result = blogic.execute(logonData);
BooleanDto dto = null;
if (result != null) {
    if (result instanceof BooleanDto) {
        dto = (BooleanDto) result;
    }
}
if (dto == null || dto.getResult() == null || !dto.getResult()) {
    throw new ServiceException("MSG0015");
}
```

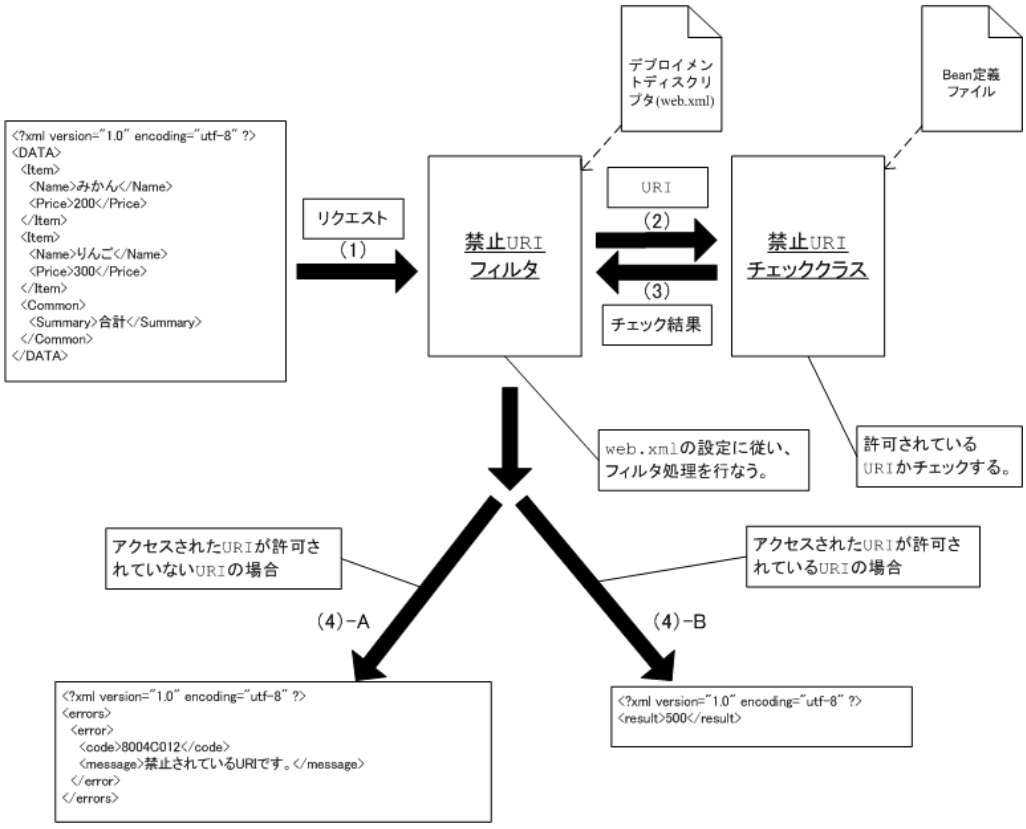
RE-01 URIアクセス制御機能

■ 概要

◆ 機能概要

- クライアントからのリクエストの任意の URI 以外へのアクセスを禁止する機能である。

◆ 概念図



◆ 解説

(1) クライアントからリクエストをWebAPサーバに送信する。
(2) フィルタはリクエストに設定されているURIを引数に、禁止URIチェッククラスを実行する。
(3) 禁止URIチェッククラスは、(2)で受け取ったURIが許可されたURIか判定し、結果を返却する。

許可するURIの一覧はBean定義ファイルに記述されている。

(4)－A (3)で返却された結果が偽だった場合、ForbiddenURIExceptionをスローする。デプロイメントディスクリプタ（web.xml）にForbiddenURIException に対応した<error-page>要素が設定されている場合、設定に従って遷移する。（デフォルトでは、“/secure/exception.do”に遷移し、フレームワークの例外処理を行う）

(4)－B (3)で返却された結果が真だった場合、アクセスを許可し、リクエストを処理する。

■ 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のコーディングポイント
  - ➢ 禁止 URI フィルタの定義
    禁止 URI フィルタ（ForbiddenURIFilter）を、デプロイメントディスクリプタ（web.xml）に定義する。
    - ✧ デプロイメントディスクリプタサンプル

```
<web-app>
  (略)
  <filter>
    <filter-name>forbiddenURIFilter</filter-name>
    <filter-class>
      jp.terasoluna.fw.web.rich.ForbiddenURIFilter
    </filter-class>
  </filter>
  <filter-mapping>
    <filter-name>forbiddenURIFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <error-page>
    <exception-type>
      jp.terasoluna.fw.web.rich.exception.ForbiddenURIException
    </exception-type>
    <location>/secure/exception.do</location>
  </error-page>
  (略)
```

同名で指定する。

禁止 URI フィルタを設定する。

禁止された URI へアクセスされた場合に、スローされる例外。

禁止された URI にアクセスしたことを示すレスポンス送信処理の URI を設定する。

- 禁止 URI チェッククラスの Bean 定義
  禁止 URI チェッククラス（ForbiddenURICheckerImpl）を定義する。
  allowedURISet 属性に、許可する URI のリストを設定する。
  - ✧ Bean 定義サンプル

```
<!-- 禁止URIチェッククラスのBean定義サンプル -->
<bean id="forbiddenURIChecker"
    class="jp.terasoluna.fw.web.rich.ForbiddenURICheckerImpl">
  <property name="allowedURISet">
    <set>
      <value>/secure/blogic.do</value>
      <value>/healthcheck.do</value>
    </set>
  </property>
</bean>
```

禁止 URI チェッククラスを設定する。

コンテキスト名を除いた形式で、許可する URI を設定する。

■ リファレンス

◆ 構成クラス

	クラス名	概要
1	jp.terasoluna.fw.web.rich.ForbiddenURIFilter	許可されている URI 以外へのアクセスを禁止するフィルタ。
2	jp.terasoluna.fw.web.rich.ForbiddenURIChecker	文字列が許可する URI か判定するチェッカのインタフェース。
3	jp.terasoluna.fw.web.rich.ForbiddenURICheckerImpl	文字列が許可する URI か判定するチェッカのデフォルト実装クラス。
4	jp.terasoluna.fw.web.rich.ForbiddenURIException	ForbiddenURIFilter で、許可されない URI に対してアクセスされたと判定された場合にスローされるクラス。

◆ 拡張ポイント

- なし

■ 関連機能

- なし

■ 使用例

- TERASOLUNA Server Framework for Java (Rich 版) 機能網羅サンプル
  - UC107 アクセス URI 制御
    - jp.terasoluna.rich.functionsample.uri.*

- TERASOLUNA Server Framework for Java (Rich 版) チュートリアル
  - 「2.7 アクセス制御」
  - /webapps/WEB-INF/web.xml
  - /webapps/WEB-INF/applicationContext.xml 等

■ 備考

- なし

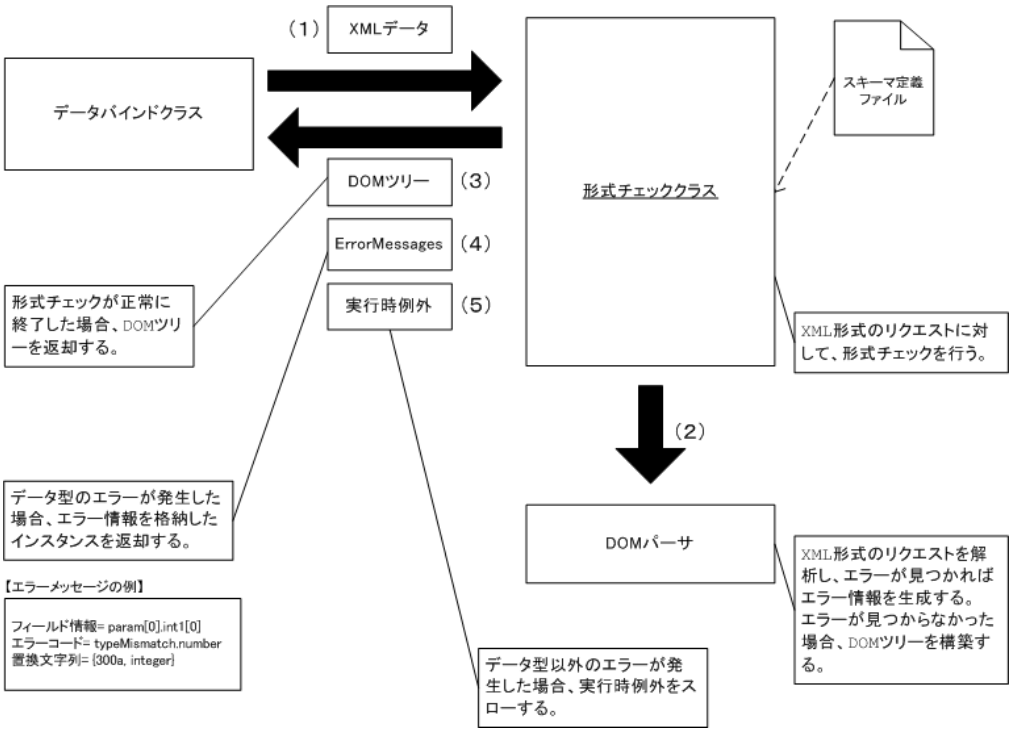
RF-01 形式チェック機能

■ 概要

◆ 機能概要

- XML 形式のリクエストデータに対し、スキーマ定義ファイルによる形式チェック（XML スキーマによる妥当性検証）を行う機能である。
- 形式チェックでは DOM パーサを利用する。
  - 形式チェックに成功した場合、DOM ツリーを返却する。
  - 形式チェックに失敗した場合、エラーハンドリングを行う。
    - データ型のエラー（リクエストデータの値が、定義されたデータ型と一致しないエラー）が発生した場合、エラー情報を生成して返却する。
    - データ型以外のエラー発生した場合、実行時例外をスローする。
- リクエストデータが XML 形式の場合のみ、XML スキーマによる形式チェックを行うことができる。
  - 形式チェックを行うメリットとして、クライアントに XML の形式エラー情報を返却することができる。
  - 形式チェックを行うデメリットとして、パフォーマンスが劣化する。
  - 形式チェックの有無は Bean 定義ファイルに設定し、システムで一意となる。
  - 上記設定を行わない場合、デフォルトで形式チェックを行う設定となる。
- XML スキーマに名前空間を宣言した場合のみ、スキーマ定義のキャッシュを行うことができる。詳細は、『コーディングポイント』を参照のこと。
  - キャッシュを行うことで、パフォーマンスを向上させることができる。
- 『RB-01 リクエストデータ解析機能』で使用される。

◆ 概念図



◆ 解説

(1) データバインダが形式チェックを実行する。
- 引数として、XML形式のリクエストデータ、バインド対象のJavaBean、エラーを格納するErrorMessagesインスタンスを設定する。

(2) DOMパーサによるリクエストデータの解析時に、XMLスキーマによる形式チェック（妥当性検証）が行われる。
- XML スキーマによる形式チェックを行うには、XML 形式のリクエストデータの構造を定義したスキーマ定義ファイルを用意する必要がある。
- スキーマ定義にマッチしない XML 形式のリクエストデータが送られてきた場合に、形式チェックのエラーが発生する。

(3) 形式チェックに成功した場合、(1)で構築されたDOMツリーをデータバインダに返却する。

(4) 形式チェックでデータ型のエラーが見つかった場合（例えば、int型で定義された要素の値に文字列が入力された場合など）、(1)で渡されたErrorMessagesインスタンスにエラー情報を格納し、データバインダに返却する。
- ErrorMessages には、フィールド情報、エラーコード、置換文字列が格納される。

(5) 形式チェックでデータ型以外のエラーが見つかった場合（例えばXML電文のタグが閉じられていない場合など)、実行時例外がスローされる。

■ 使用方法

◆ コーディングポイント

- 形式チェッククラスの Bean 定義
  形式チェックを行う場合、Bean 定義ファイルに形式チェッククラスの設定を行う。
  - ✧ 詳細は、『RB-01 リクエストデータ解析機能』を参照のこと。
- XML データのスキーマ定義
  XML 形式のリクエストデータを形式チェックする場合、スキーマ定義ファイルに設定すべきポイントとして、下記の項目が挙げられる。
  1. リクエストデータの XML の構造を定義する。
  2. スキーマ定義ファイルのファイル名を、リクエストデータのバインド先の JavaBean の“クラス名 + .xsd”にして、同じディレクトリに配置する。
     例）JavaBean が「sample.SampleDto」クラスの場合、スキーマ定義ファイルはクラスパス上の「sample/SampleDto.xsd」ファイルとなる。
  3. 名前空間を定義する。（オプション）
     スキーマ定義ファイルに名前空間を宣言することができる。（※一つのスキーマファイルに対して、一意の名前空間を宣言すること）
     名前空間を宣言した場合のみ、スキーマ定義ファイルのキャッシュを行なうことが可能となる。
     名前空間を宣言した場合、以下のファイルに設定を行う必要がある。
     - ✧ JavaBean と名前空間の関連付けを行うプロパティファイル
       名前空間を宣言する場合、クラスパス上に配置した設定ファイル（namespaces.properties）にリクエストデータのバインド先の JavaBean と名前空間の関連付けを行う必要がある。詳細は後述の『コーディングポイント』を参照のこと。
     - ✧ マッピング定義ファイル
       対応するマッピング定義ファイルにも名前空間を宣言する必要がある。
       詳細は、『RB-03 XML-Object 変換機能』を参照のこと。
     - ✧ Bean 定義ファイル
       形式チェッククラスの初期化処理を行う必要がある。
       詳細は、『RB-01 リクエストデータ解析機能』を参照のこと。

- 名前空間を宣言しない場合のサンプル
  - ✧ XML データサンプル
    リクエストデータとして送信される XML データのサンプルを以下に記す。

```
<sample-dto>
  <user-id>0001</user-id>
  <user-name>satou</user-name>
  <item>
    <id>123</id>
    <name>item1</name>
    <price>300</price>
  </item>
  <item>
    <id>124</id>
    <name>item2</name>
    <price>100</price>
  </item>
  <member>true</member>
  <registerDate>2006-03-15T12:57:30</registerDate>
</sample-dto>
```

ルート要素。

item 要素は子要素を持つので、データ型（複合型）で宣言する必要がある。

✧ スキーマ定義ファイルサンプル

上記 XML データサンプルに対応するスキーマ定義ファイルを生成する。

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="sample-dto" type="sample-dto-type"/>

  <xs:complexType name="sample-dto-type">
    <xs:sequence>
      <xs:element name="user-id" type="xs:int" />
      <xs:element name="user-name" type="xs:string" />
      <xs:element name="item" type="item-type"
          maxOccurs="unbounded" />
      <xs:element name="member " type="xs:boolean" />
      <xs:element name="registerDate" type="xs:dateTime" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="item-type">
    <xs:sequence>
      <xs:element name="id" type="xs:int" />
      <xs:element name="name" type="xs:string" />
      <xs:element name="price" type="xs:int" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

ルート要素を宣言。

ルート要素のデータ型（複合型）を定義する。

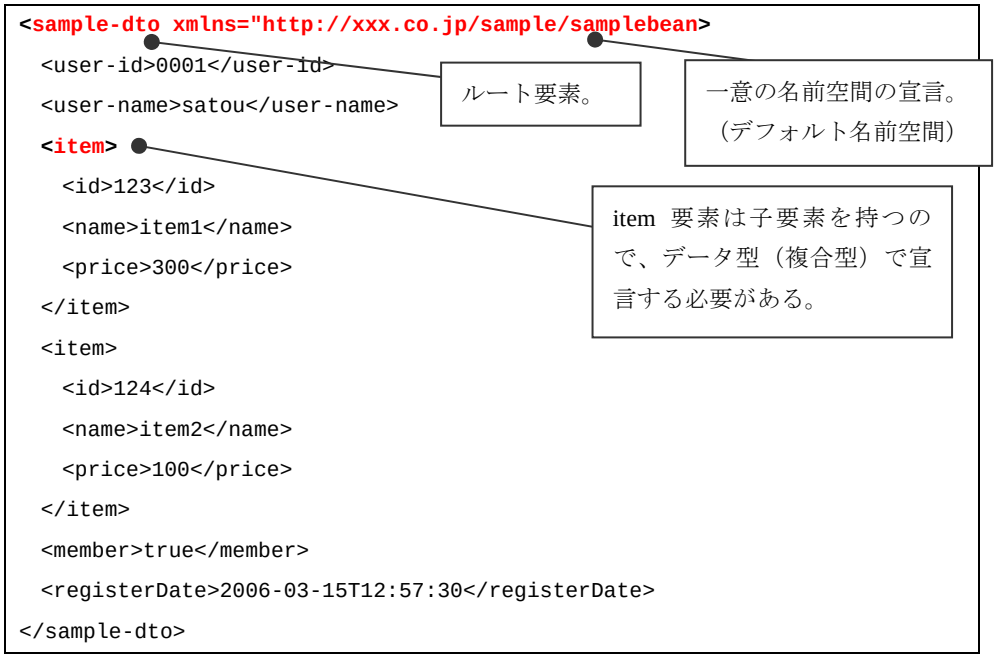
繰り返し要素の場合は、maxOccurs 属性に繰り返し数を定義する。
※unbounded は繰り返し数が無制限

item 要素のデータ型（複合型）を定義する。

- 名前空間を宣言する場合のサンプル
  - XML データサンプル

    リクエストデータとして送信される XML データのサンプルを以下に記す。

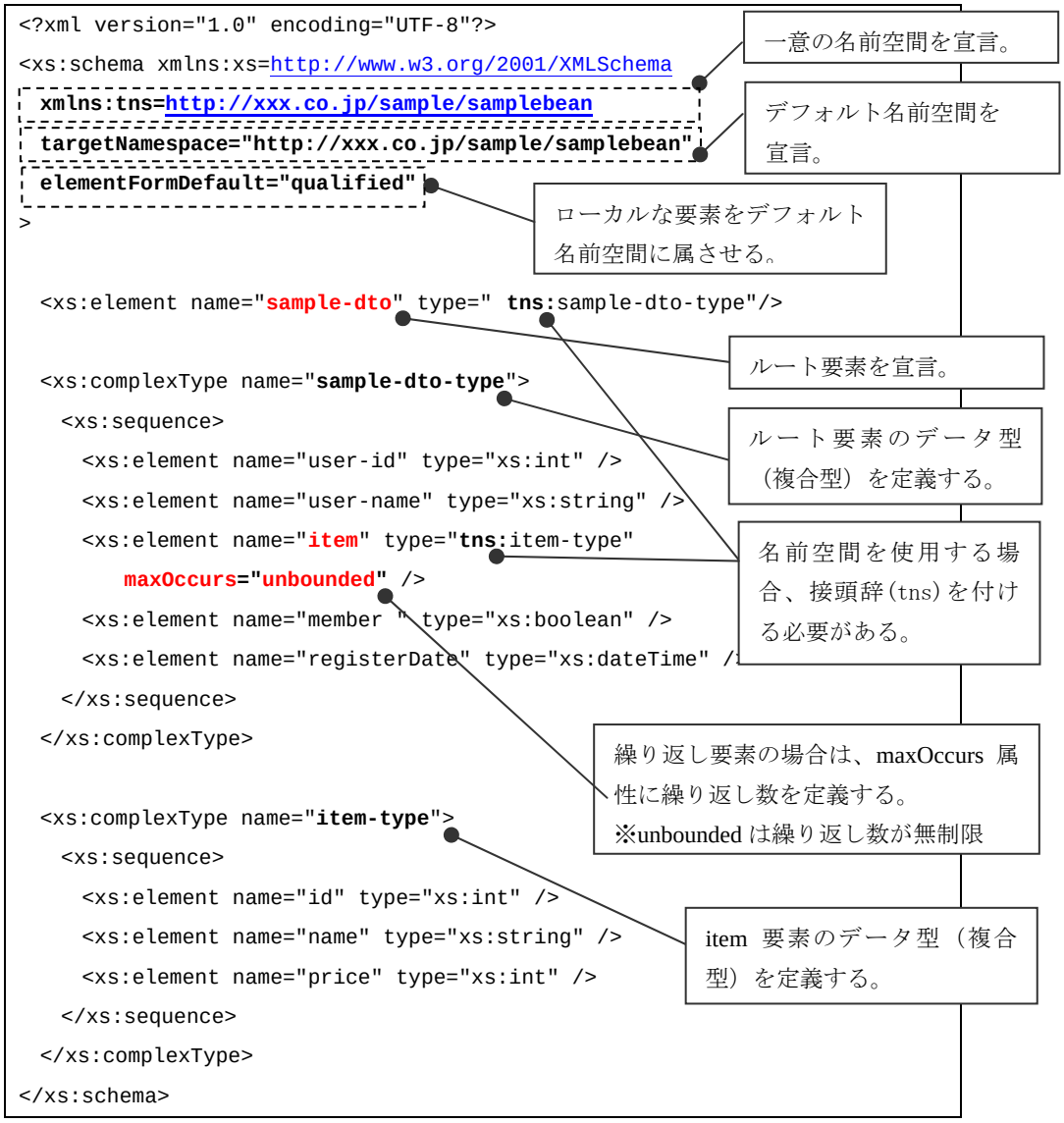
    ※必ず 1 つの JavaBean に対し、1 つの名前空間を設定すること。




```
<sample-dto xmlns="http://xxx.co.jp/sample/samplebean">
  <user-id>0001</user-id>
  <user-name>satou</user-name>
  <item>
    <id>123</id>
    <name>item1</name>
    <price>300</price>
  </item>
  <item>
    <id>124</id>
    <name>item2</name>
    <price>100</price>
  </item>
  <member>true</member>
  <registerDate>2006-03-15T12:57:30</registerDate>
</sample-dto>
```

✧ スキーマ定義ファイルサンプル

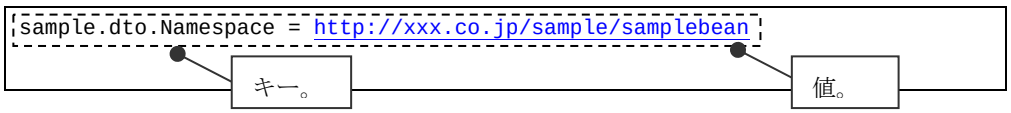
上記 XML データサンプルに対応するスキーマ定義ファイルを生成する。



✧ 名前空間定義ファイルサンプル

プロパティファイル（namespaces.properties）にリクエストデータのバインド先の JavaBean と名前空間の関連付けを記述し、クラスパス上に配置する。

➢ キーは「JavaBean のパッケージ名 + クラス名 + .“Namespace”」、値は名前空間となる。



■ リファレンス

◆ 構成クラス

	クラス名	概要
1	SchemaValidator	XML データの形式チェックを行うクラスが実装すべきインタフェース。
2	SchemaValidatorImpl	XML データの形式チェックを行う SchemaValidator デフォルト実装クラス。
3	ErrorMessages	形式チェックで発生したエラーメッセージが格納されるクラス。
4	DOMParser	XML 形式のリクエストデータのパース(解析)を行い、DOM ツリーを構築するクラス。 パース時に XML スキーマによる形式チェックを行う。
5	XMLErrorReporterEx	詳細なエラー情報を取得するために、Xerces のエラーレポート機能を拡張したクラス。

◆ XML Schemaがサポートするデータ型

- <element>要素の type 属性に記述する。

	名前	対応する JavaBean のデータ型
1	string	java.lang.String
2	int	int or java.lang.Integer
3	long	long or java.lang.Long
4	boolean	boolean or java.lang.Boolean
5	double	double or java.lang.Double
6	float	float or java.lang.Float
7	decimal	java.math.BigDecimal
8	byte	byte or java.lang.Byte
9	dateTime	java.util.Date
10	short	short or java.lang.Short
11	char	char or java.lang.Character

※通常使用しない定義は省略している。詳細は XML Schema のドキュメントを参照すること。

◆ エラーコード

- データ型の形式チェックエラーが発生した場合に、返却されるエラーコードの一覧を記す。

エラーコード	置換文字列	概要
typeMismatch.number	{要素名，データ型，入力値}	定義されたデータ型(数値)と異なるデータ型が入力された場合 数値型の形式：int、long、boolean、double、float、decimal、byte、dateTime、、short、char。
typeMismatch.boolean	{要素名，入力値}	定義された boolean 型と異なるデータ型が入力された場合 boolean 型の形式:true/false or 0/1
typeMismatch.date	{要素名，入力値}	定義された日付型(date,dateTime,time)と異なるデータ型が入力された場合 date 型の形式:YYYY-MM-DD dateTime 型の形式: YYYY-MM-DDThh:mm:ss time 型の形式:hh:mm:ss.sss ※Castor を用いて JavaBean の date 型の属性にマッピングする場合は、dateTime 型で定義すること。
typeMismatch.numberMaxRange	{要素名，データ型の最大値，データ型，入力値}	定義されたデータ型(数値)の最大値以上の数値が入力された場合 数値型の形式については、上記 typeMismatch.number **の概要を参照のこと。**
typeMismatch.numberMinRange	{要素名，データ型の最小値，データ型，入力値}	定義されたデータ型(数値)の最小値以下の数値が入力された場合 数値型の形式については、上記 typeMismatch.number **の概要を参照のこと。**

◆ 拡張ポイント

- SchemaValidatorImpl クラスの getUrl()メソッドをオーバーライドすることで、スキーマ定義ファイルの取得方法を変更することができる。

■ 関連機能

- 『RB-01 リクエストデータ解析機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Rich 版) **機能網羅サンプル**
  - UC108 **形式チェック**

- ✧ jp.terasoluna.rich.functionsample.formcheck.*

- TERASOLUNA Server Framework for Java (Rich 版) チュートリアル
  - 「2.5.1 電文形式チェック」
  - jp.terasoluna.rich.tutorial.service.bean.UserBean.java
  - jp.terasoluna.rich.tutorial.service.bean.UserBean.xsd
  - /sources/namespaces.properties 等

■ 備考

- **本機能を利用する場合の注意事項**

  本機能を利用する場合、極端に性能が劣化するため、なるべく利用しないことを推奨とします。本機能によりXMLスキーマに基づいて、XML自体の文法をチェックすることができますが、これとは別に要素に格納された入力データ自身の検証を行う必要があります。性能を重視する場合には、本機能による形式チェックは省略し、入力チェック機能などを用いた入力データ自身の検証に重点を置くことをお勧めします。

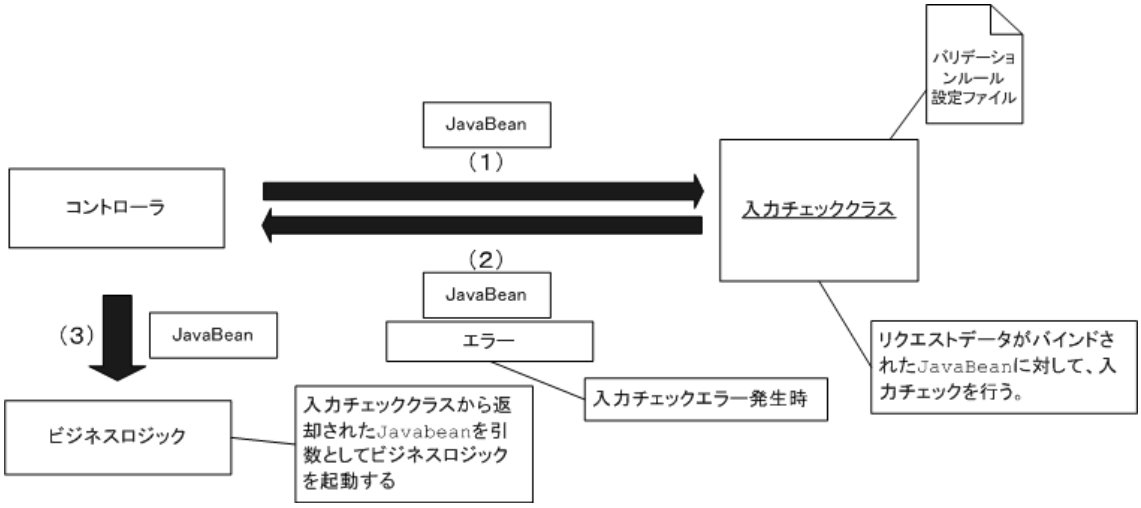
RF-02 入力チェック機能

■ 概要

◆ 機能概要

- クライアントにて入力された値が妥当であるかをチェックする機能である。
  - 『RB-01 リクエストデータ解析機能』で、JavaBean にバインドされたリクエストデータの値を検査する。
- 以下のチェックが可能である。
  - 設定ファイルによる入力チェック
    - ✧ 単項目チェック
      チェック対象の JavaBean 内の 1 属性に対して、1 種類のルールによるチェックを行う。
    - ✧ 1 属性・2 種類以上の入力チェック
      チェック対象の JavaBean 内の 1 属性に対して、複数のルールによるチェックを行う。
    - ✧ 配列・コレクション型への入力チェック
      チェック対象の JavaBean 内の配列・コレクション型の 1 属性に対して、チェックを行う。
    - ✧ JavaBean1 クラス・2 パターンの入力チェック
      チェック対象の JavaBean に対して、条件に応じ、異なる内容のチェックを行う。
  - コーディングによる入力チェック
    提供されているチェックルールを使用せず、プロジェクトが作成した Java コードによるチェックを行う。
- 提供ルール（単項目チェック）
  TERASOLUNA Server Framework for Java より提供している入力チェックは、必須チェック、半角カナ文字列チェック、禁止文字列チェックなど 28 種類ある。詳細は後述の『入力チェックルール解説』を参照のこと。

◆ 概念図



◆ 解説

(1) コントローラは受信したリクエストデータがバインドされたJavaBeanを入力チェッククラスに渡す。
(2) 入力チェッククラスはバリデーションルール定義ファイル（validator-rules.xml）に定義されたチェックルールに従い、入力チェックを行い、結果をJavaBeanとして返す。エラー発生時にはエラーを格納する。（詳細は『RD-01 例外ハンドリング機能』を参照のこと。）
(3) コントローラは(2)で返却されたJavaBeanを引数に、ビジネスロジックを起動する（詳細は『RA-02 コントローラ拡張機能』を参照のこと）。

■ 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
  TERASOLUNA Server Framework for Java (Rich 版)では Commons-Validator との連携を行う Spring-modules を利用した実装を提供している。
  - Spring-modulesとは、Springフレームワークの拡張モジュールである。本機能では、その中のspringmodules-validatorを利用している。
    (https://springmodules.dev.java.net/)
  - バリデータの Bean 定義
    入力チェックを行うバリデータと、バリデータのファクトリクラスを Bean 定義する。
    - Bean 定義サンプル

```
<!--- Spring-Modules Validatorファクトリのサンプルbean定義 -->
<bean id="validatorFactory"
    class="org.springmodules.commons.validator.DefaultValidatorFactory">
  <property name="validationConfigLocations">
    <list>
      <value>/WEB-INF/validation/validator-rules.xml</value>
      <value>/WEB-INF/validation/validator-rules-ex.xml</value>
      <value>/WEB-INF/validation/validation.xml</value>
    </list>
  </property>
</bean>

<!-- バリデータのサンプルBean定義 -->
<bean id="beanValidator"
    class="org.springmodules.commons.validator.DefaultBeanValidator">
  <property name="validatorFactory" ref="validatorFactory"/>
</bean>
```

入力チェック機能のルールを定義したファイルを設定する。

バリデーション定義ファイルを設定する。後述の『業務開発者のコーディングポイント』を参照のこと。

バリデータとルールファイルを関連付ける。

✧ 入力チェックのルール定義

バリデーションルール定義ファイルに、入力チェックのルールを定義する。

※TERASOLUNA Server Framework for Java (Rich 版)は標準でバリデーションルール定義ファイル（validator-rules.xml、validator-rules-ex.xml）を提供する。

➢ バリデーションルール定義ファイル(validator-rules.xml)

単項目の入力チェックで使用する、入力チェックルールの定義。TERASOLUNA Server Framework for Java (Rich 版) より提供されている。

```
……
<!-- 必須入力 -->
<validator name="required"
    classname="jp.terasoluna.fw.validation.FieldChecks"
    method="validateRequired"
    methodParams="java.lang.Object,
    org.apache.commons.validator.ValidatorAction,
    org.apache.commons.validator.Field,
    jp.terasoluna.fw.validation.ValidationErrors"
    msg="errors.required"/>
</validator>
……
```

業務開発者の利用するルール名を指定する。

➢ バリデーションルール定義ファイル(validator-rules-ex.xml)

配列・コレクションの入力チェックで使用する、入力チェックルールの定義。TERASOLUNA Server Framework for Java (Rich 版) より提供されている。

```
……
<!-- 必須入力 配列・コレクション用-->
<validator name="requiredArray"
    classname="jp.terasoluna.fw.validation.FieldChecks"
    method="validateArraysIndex"
    methodParams="java.lang.Object,
    org.apache.commons.validator.ValidatorAction,
    org.apache.commons.validator.Field,
    jp.terasoluna.fw.validation.ValidationErrors"
    msg="errors.required"/>
</validator>
……
```

業務開発者の利用するルール名を指定する。

- 入力チェックエラー時のメッセージ設定。
  メッセージリソースサンプル（error-messages.properties.sjis）
  以下のようにエラーコードとメッセージが定義されていることとする。

```
errors.required = {0}は入力必須項目です.
```

- メッセージリーソースの設定。
  プロパティファイルからメッセージを取得する。（複数ファイル可）
  以下に messageSource の Bean 定義設定例を記す。

```
<!-- メッセージの設定 -->
<bean id="messageSource“
   class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames“ value=“applicationResources,error-messages”/>
</bean>
```

プロパティファイルを指定する。

- 業務開発者が行うコーディングポイント
  入力チェックのパターンにより、コーディング方法が変わる。以下の 5 パターンの入力チェックがある。
  - 設定ファイルによる入力チェック
    - 単項目チェック
    - 1 属性・2 種類以上の入力チェック
    - 配列・コレクション型への入力チェック
    - JavaBean1 クラス・2 パターンの入力チェック
  - コーディングによる入力チェック

  以下に各パターンについての設定方法を説明する。

- 設定ファイルによる入力チェック
  - ✧ 単項目チェックの設定

    チェック対象の JavaBean 内の 1 属性に対して、1 種類の入力チェックを実施する。
    - バリデーション定義ファイルの設定

      バリデーション定義ファイルにチェック対象の JavaBean 名と、チェックする属性、入力チェックのルール名を設定する。
      - サンプル

        ValidateSampleController の引数 SampleBean に対して、入力チェックを行う場合のサンプルを以下に記す。

        ✧ JavaBean サンプル（SampleBean）

```
public class SampleBean {
  private String id;
  private Item item;
  ……（get,setメソッドなどは省略）

}

public class Item {
  private String itemName;
  ……（get,setメソッドなどは省略）
}
```

ネストされた JavaBean。

✧ SampleBean の属性に対して必須入力チェックを行う場合の、バリデーション定義ファイルサンプル（validation.xml）

バリデーション定義ファイルの form 要素の name に sampleBean を、field 要素の property 属性に id を、depends に required（必須入力チェック）を指定する。

```
……
  <form name="sampleBean">
    <field property="id" depends="required">
      <arg key="field1" position="0"/>
    </field>
  </form>
……
```

チェック対象の JavaBean のクラス名を指定する。る。頭文字は小文字より開始する。
ただし、複数の文字が存在し、先頭の文字と 2 番目の文字がどちらも大文字である場合は、大文字のまま指定する。
※完全修飾名を設定する場合には、Bean 定義 にプロパティ useFullyQualifiedClassName を追加し、 true を設定する

sampleBean の属性 id。

必須入力チェック。指定可能な文字列は、下記、単項目チェックで利用できる入力チェックルール一覧を参照のこと。

✧ SampleBean の属性 item（ネストされた JavaBean）内の属性 itemName に対して必須入力チェックを行う場合の、バリデーション定義ファイルサンプル(validation.xml)
バリデーション定義ファイルの form 要素の name 属性に sampleBean を、field 要素の property 属性に item.itemName を、depends に required（必須入力チェック）を指定する。

```
.....
  <form name="sampleBean">
    <field property="item.itemName" depends="required">
      <arg key="field1" position="0"/>
    </field>
  </form>
.....
```

ネストされた JavaBean 名

SampleBean の item 属性内の itemName 属性

必須入力チェック。指定可能な文字列は、下記、単項目チェックで利用できる入力チェックルール一覧を参照のこと。

➢ ValidateSampleController のサンプル Bean 定義

```
<!-- コントローラのサンプルBean定義 -->
<bean name="/validateSampleController"
    class="jp.terasoluna.sample2.web.controller.ValidateSampleController"
    parent="xmlRequestController">
  <property name="sumService" ref="sumService"/>
  <property name="validator" ref="beanValidator"/>
</bean>
```

ソフトウェアアーキテクトが Bean 定義したバリデータを指定する。

- TERASOLUNA Server Framework for Java (Rich 版)が提供する単項目チェック一覧
  設定方法の詳細は、後述『入力チェックルール解説』を参照のこと。

ルール名	概要
required	必須入力チェック
mask	正規表現一致チェック
byte	byte 型チェック
short	short 型チェック
integer	int 型チェック
long	long 型チェック
float	float 型チェック
double	double 型チェック
date	日付形式チェック
intRange	int 型範囲チェック
doubleRange	double 型範囲チェック
floatRange	float 型範囲
maxLength	最大文字数制限
minLength	最小文字数制限
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	大文字半角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
byteRange	byte 列長範囲チェック
dateRange	date 型範囲チェック
arrayRange	配列要素数チェック
url	URL 形式チェック

✧ 1 属性・2 種類以上の入力チェックの設定

チェック対象の JavaBean 内の 1 属性に対して、2 種類以上の入力チェックを実施する。入力チェックエラー時の動作により設定方法が異なる。

- 入力チェックエラー時、次のチェックを行わない場合。
  エラーが発生した場合は同じ属性内の次のチェックは行わない。
  例えば、下記のサンプルで required チェックにおいて、エラーが発生した場合は、次のチェックである long チェックを行わない。
  - ➢ チェック対象属性とチェックルールの関連付け
    - ✧ バリデーション定義ファイルサンプル(validation.xml)

```
……
  <form name=“sampleBean">
    <field property=“field1" depends="required,long">
      <arg key="field1" position="0"/>
    </field>
  </form>
……
```

カンマ(,)区切りで入力チェックを追加する。

- 入力チェックエラー時でも全てのチェックを行う場合。
  エラーが発生した場合でも設定されている全てのチェックを行う。
  例えば、下記のサンプルで required チェックにおいて、エラーが発生した場合でも、続けて long チェックを行う。
  - ➢ チェック対象属性とチェックルールの関連付け
    - ✧ バリデーション定義ファイルサンプル(validation.xml)

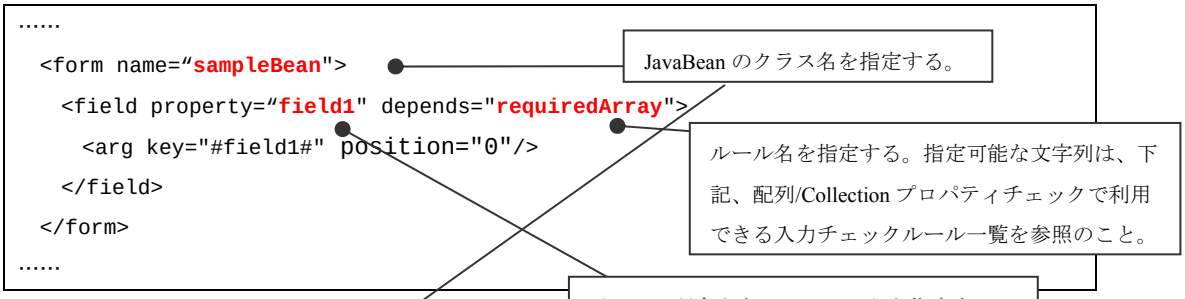
```
……
  <form name=“sampleBean">
    <field property=“field1" depends="required">
      <arg key="field1" position="0"/>
    </field>
    <field property=“field1" depends="long">
      <arg key="field1" position="0"/>
    </field>
  </form>
……
```

同フィールドに対して入力チェックをそれぞれ設定する。

- コントローラへの入力チェック機能の設定、及び利用可能なルールは、『単項目チェックの設定』に従う。

✧ 配列・コレクション型への入力チェックの設定
チェック対象の JavaBean 内の配列・コレクション型の 1 属性に対して、チェックを行う。チェックしたい JavaBean と利用したい入力チェックルール名を指定する。また、チェック時に呼び出すバリデータの指定をする。

- チェック対象属性とチェックルールの関連付け。
  - ➢ バリデーション定義ファイルサンプル(validation.xml)



  - ➢ チェック対象の JavaBean サンプル(getter,setter は省略)

```
public class SampleBean {
    private String[] field1;
    ……(get,setメソッドなどは省略)
}
```

- 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。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 commons-validator のリファレンスを参照のこと。

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

- コントローラへの入力チェック機能の設定は、単項目チェックの設定に従う。

- ✧ JavaBean 1 クラス・2 パターンの入力チェックの設定
  - 複数リクエストに対して同一の Bean を紐付ける場合、各々のリクエストで異なるパターンの入力チェックを利用する際は、後述の『備考 その他の入力チェック』を参照のこと。

➢ コーディングによる入力チェックの設定

Java によるコーディングで作成したプロジェクト独自の入力チェッククラスを用いて、入力チェックを行う。主に、項目間の相関チェックなど、複雑な入力チェックを行う場合に使用する。前述の設定ファイルによる単項目チェックとの併用も可能である。

※ただし、独自の入力チェッククラスのみを使用する場合でも validation.xml の form 定義に空の定義が必要である。

✧ 入力チェッククラスの生成

JavaBean に対しての入力チェックを行う BaseMultiFieldsValidator 継承クラスを生成する。

● 入力チェッククラスサンプル

field1 と field2 の相関チェック。field1 が null の場合、field2 は必須。field2 が null の場合、field1 が必須であることを相関的にチェックする。

```
public class SampleMultiFieldsValidator extends BaseMultiFieldsValidator {
    /**
     * 相関チェックを行なうメソッド
     */
    @Override
    protected void multiFieldsValidate(Object obj, Errors errors) {
        TestBean bean = (TestBean) obj;
        // JavaBeanから値を取り出す
        String field1 = bean.getField1();
        String field2 = bean.getField2();
        //チェックロジックを記述
        boolean bool = true;
        if (field1 == null) {
            // feild1がnullならば、feild2は必須。
            if (field2 == null) {
                bool = false;
            // feild1がnullでないならば、feild2はnull。
            }
        } if (field2 != null) {
            bool = false;
        }
        // チェックがfalseならば、エラーを追加する
        if (!bool) {
            errors.rejectValue("name", "errors.sample");
        }
    }
}
```

BaseMultiFieldsValidator を継承して、実装する。

単項目チェック後に実行されるメソッド

✧ 入力チェッククラスの Bean 定義

● Bean 定義サンプル

上記で生成した入力チェッククラスの Bean 定義を行う。

```
<!- 入力チェッククラスのBean定義サンプル-->
<bean id="sampleValidator"
    class="jp.terasoluna.sample2.validation.SampleMultiFieldsValidator">
  <property name="validatorFactory" ref="validatorFactory"/>
</bean>
```

生成した入力チェッククラス名を指定する。

✧ コントローラの Bean 定義

コントローラの validator 属性に、入力チェッククラスを設定する。

● Bean 定義サンプル

上記で生成した入力チェッククラスを用いてチェックを行う場合の Bean 定義を以下に記す。

```
<!-- コントローラのBean定義サンプル -->
<bean name="/validateSampleController"
    class="jp.terasoluna.sample2.web.controller.ValidateSampleController"
    parent="xmlRequestController">
  <property name="sumService" ref="sumService"/>
  <property name="validator" ref="sampleValidator"/>
</bean>
```

生成した入力チェッククラスを設定する。

● 入力チェックの型について

設定ファイルによる入力チェックは、基本的に String 型を対象としている。以下に String 型以外の型に対する入力チェックの注意点を紹介する。

➢ プリミティブ型に対する必須入力チェック(required)はできない。
(プリミティブ型の属性は初期化されるため)

➢ Date 型に対する日付形式チェック(date)、date 型範囲チェック(dateRange)を行うことはできない。

✧ Date 型に対する date 型チェック、date 型範囲チェックを行う場合は、コーディングによる入力チェックを行う、または日付を文字列として入力する。(※Date 型に対する date 型チェックを行う意味はない)

■ リファレンス

◆ 入力チェックルール解説

以下に単項目入力チェックの代表的なチェックルールを紹介する。詳細な内容については FieldChecks クラスの JavaDoc を参照すること。

- required

入力値が Null ではないかどうかのチェックと、スペースを除いた入力値の文字列長が 0 より大きいか チェックする。

➢ バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="userName" depend="required">
    <arg key="userName" position="0"/>
  </field>
</form>
……
```

デフォルトのメッセージは「errors.required={0}は入力必須項目です.」

- mask

  入力値が指定された正規表現に適合するかチェックする。

  ➢ バリデーション設定ファイル（validation.xml）に設定を要する<var>要素

var-name	var-value	必須	概要
mask	正規表現	○	入力文字列が指定された正規表現に合致するときは true が返却される。指定されない場合は ValidatorException がスローされる。

  ➢ バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="userName" depends="mask">
    <arg key="userName" position="0"/>
    <var>
      <var-name>mask</var-name>
      <var-value>^([0-9a-zA-Z])*$</var-value>
    </var>
  </field>
</form>
……
```

デフォルトのメッセージは「errors.mask={0}の入力形式が間違っています。」

- date

  入力値が有効な日付かチェックする。

➢ バリデーション設定ファイル（validation.xml）に設定を要する<var>要素

var-name	var-value	必須	概要
datePattern	日付パターン	×	日付パターンを指定する。入力値の桁数チェックは行わない。 たとえば、日付パターンが yyyy/MM/dd の場合、2001/1/1 はエラーにならない。 datePattern と datePatternStrict が両方指定されている場合は、 datePattern が優先して使用される。
datePatternStrict	日付パターン	×	日付パターンを指定する。入力値の桁数が、 指定された日付パターンの桁数に合致するかのチェックを行う。 たとえば、日付パターンが yyyy/MM/dd の場合、2001/1/1 はエラーになる。 datePattern と datePatternStrict が両方指定されている場合は、 datePattern が優先して使用される。

➢ バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="dateField" depends="date">
    <arg key="dateField" position="0"/>
    <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.date={0}には正しい日付を入力してください.」

- intRange

  入力値が int 型に変換可能であり、 かつ指定された範囲内か検証する。

  ➢ バリデーション設定ファイル（validation.xml）に設定を要する<var>要素

var-name	var-value	必須	概要
intRangeMin	最小値	×	範囲指定の最小値を設定する。設定しない場合、Integer の最小値が 指定される。 数値以外の文字列が入力された場合、例外がスローされる。
intRangeMax	最大値	×	範囲指定の最大値を設定する。設定しない場合、Integer の最大値が 指定される。 数値以外の文字列が入力された場合、例外がスローされる。

  ➢ バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="intField" depends="intRange">
    <arg key="intField" position="0"/>
    <arg key="${var:intRangeMin}" position="1"/>
    <arg key="${var:intRangeMax}" position="2"/>
    <var>
      <var-name>intRangeMin</var-name>
      <var-value>10</var-value>
    </var>
    <var>
      <var-name>intRangeMax</var-name>
      <var-value>100</var-value>
    </var>
  </field>
</form>
……
```

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

- doubleRange

  入力値が double 型に変換可能であり、 かつ指定された範囲内か検証する。

  ➢ バリデーション設定ファイル（validation.xml）に設定を要する<var>要素

var-name	var-value	必須	概要
doubleRangeMin	最小値	×	範囲指定の最小値を設定する。設定しない場合、Double 型の最小値が 指定される。 数値以外の文字列が入力された場合、例外がスローされる。
doubleRangeMax	最大値	×	範囲指定の最大値を設定する。設定しない場合、Double 型の最大値が 指定される。 数値以外の文字列が入力された場合、例外がスローされる。

  ➢ バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="doubleField" depends="doubleRange">
    <arg key="doubleField" position="0"/>
    <arg key="${var:doubleRangeMin}" position="1"/>
    <arg key="${var:doubleRangeMax}" position="2"/>
    <var>
      <var-name>doubleRangeMin</var-name>
      <var-value>10</var-value>
    </var>
    <var>
      <var-name>doubleRangeMax</var-name>
      <var-value>100</var-value>
    </var>
  </field>
</form>
……
```

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

- floatRange

入力値が float 型に変換可能であり、 かつ指定された範囲内か検証する。

➢ バリデーション設定ファイル（validation.xml）に設定を要する<var>要素

var-name	var-value	必須	概要
floatRangeMin	最小値	×	範囲指定の最小値を設定する。設定しない場合、Float 型の最小値が 指定される。 数値以外の文字列が入力された場合、例外がスローされる。
floatRangeMax	最大値	×	範囲指定の最大値を設定する。設定しない場合、Float 型の最大値が 指定される。 数値以外の文字列が入力された場合、例外がスローされる。

➢ バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="floatField" depends="floatRange">
    <arg key="floatField" position="0"/>
    <arg key="${var:floatRangeMin}" position="1"/>
    <arg key="${var:floatRangeMax}" position="2"/>
    <var>
      <var-name>floatRangeMin</var-name>
      <var-value>10</var-value>
    </var>
    <var>
      <var-name>floatRangeMax</var-name>
      <var-value>100</var-value>
    </var>
  </field>
</form>
……
```

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

- maxLength

  入力値の文字数が指定された最大文字数以下かチェックする。

➢ バリデーション設定ファイル（validation.xml）に設定を要する<var>要素

var-name	var-value	必須	概要
maxlength	最大文字数	○	文字列の最大文字数を設定する。 数値以外の文字列が入力された場合、例外がスローされる。

➢ バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="userName" depends="maxLength">
    <arg key="userName" position="0"/>
    <arg key="${var:maxlength}" position="1"/>
    <var>
      <var-name>maxlength</var-name>
      <var-value>10</var-value>
    </var>
  </field>
</from>
……
```

デフォルトのメッセージは「errors.maxlength={0}には{1}文字以下で入力してください.」

- minLength

  入力値の文字数が指定された最小文字数以上かチェックする。

  ➢ バリデーション設定ファイル（validation.xml）に設定を要する<var>要素

var-name	var-value	必須	概要
minlength	最小文字数	○	文字列の最小文字数を設定する。 数値以外の文字列が入力された場合、例外がスローされる。

  ➢ バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="userName" depends="minLength">
    <arg key="userName" position="0"/>
    <arg key="${var:minlength}" position="1"/>
    <var>
      <var-name>minlength</var-name>
      <var-value>10</var-value>
    </var>
  </field>
<from>
……
```

デフォルトのメッセージは「errors.minlength={0}には{1}文字以上で入力してください.」

- 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。

  ➢ バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="userName" depend="alphaNumericString">
    <arg key="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
  入力文字列がいわゆる半角カナ文字のみであることをチェックする。
  - バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="userName" depend="hankakuKanaString">
    <arg key="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。
- ｱｲｳｴｵｧｨｩｪｫｶｷｸｹｺｻｼｽｾｿﾀﾁﾂｯﾃﾄﾅﾆﾇﾈﾉﾊﾋﾌﾍﾎﾏﾐﾑﾒﾓﾔﾕﾖｬｭｮﾗﾘﾙﾚﾛﾜﾝﾞﾞｰ･、。「」

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サンプル

```
validation.hankaku.kana.list=ｱｲｳｴｵｧｨｩｪｫｶｷｸｹｺｻｼｽｾｿﾀﾁﾂｯﾃﾄﾅﾆﾇﾈﾉﾊﾋﾌﾍﾎﾏﾐﾑﾒﾓﾔﾕﾖｬｭｮﾗﾘﾙﾚﾛﾜﾝﾞﾞ
```

- hankakuString
  入力文字列がいわゆる半角文字のみであることをチェックする。
  ➢ バリデーション設定ファイルサンプル（validation.xml）

```
.....
<form name="validate">
  <field property="userName" depend="hankakuString">
    <arg key="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。
  ➢ バリデーション設定ファイルサンプル（validation.xml）

```
.....
<form name="validate">
  <field property="userName" depend="zenkakuString">
    <arg key="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。
  - バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="userName" depend="zenkakuKanaString">
    <arg key="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。
  - アイウヴエオァィゥェォカキクケコヵ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ヮヰヱヲッンー

  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サンプル

```
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。
  - バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="userName" depend="capAlphaNumericString">
    <arg key="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。
  - バリデーション設定ファイル（validation.xml）に設定を要する<var>要素

var-name	var-value	必須	概要
integerLength	整数部桁数	×	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、検証を行なわない。
scale	小数部桁数	×	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、検証を行なわない
isAccordedInteger	整数桁数一致チェック	×	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	×	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。

  - バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="numberField" depend="number">
    <arg key="numberField" 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。
  - バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="numericField" depend="numericString">
    <arg key="numericField" 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。
  - バリデーション設定ファイル（validation.xml）に設定を要する<var>要素

var-name	var-value	必須	概要
chars	入力禁止キャラクタ	×	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。

  - バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="userName" depend="prohibited">
    <arg key="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

- stringLength
  入力文字列の桁数をチェックする。
  - バリデーション設定ファイル（validation.xml）に設定を要する<var>要素

var-name	var-value	必須	概要
stringLength	入力文字列桁数	×	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。

  - バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="userName" depend="stringLength">
    <arg key="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

- 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。
  - バリデーション設定ファイル（validation.xml）に設定を要する<var>要素

var-name	var-value	必須	概要
maxByteLength	最大バイト数	×	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。
minByteLength	最小バイト数	×	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	×	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。

  - バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="userName" depend="byteRange">
    <arg key="label.userName" position="0" />
    <arg key="${var:minByteLength}" position="1" resource="false" />
    <arg key="${var:maxByteLength}" position="2" resource="false" />
    <var>
      <var-name>maxByteLength</var-nam
      <var-value>16</var-value>
    </var>
    <var>
      <var-name>minByteLength</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}のバイト数は{1}バイトから{2}バイトの範囲にしてください.」

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

- 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。
  - バリデーション設定ファイル（validation.xml）に設定を要する<var>要素

var-name	var-value	必須	概要
startDate	開始日付	×	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	×	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	×	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	×	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。

  - バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="dateField" depend="dateRange">
    <arg key="dateField" 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}の日付は{1}から{2}の範囲にしてください.」

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。

- url
  指定された属性が URL 形式であることをチェックする。
  - バリデーション設定ファイル（validation.xml）に設定を要する<var>要素

var-name	var-value	必須	概要
allowAllSchemes	真偽	×	全てのスキームを許可するかの判断するフラグ。デフォルト(false)では許可をしない。許可した場合は schemes で設定した値は無視される。
allow2Slashes	真偽	×	ダブルスラッシュ(//)を許可するか判断するフラグ。デフォルト(false)では許可をしない。
noFragments	真偽	×	URL が分割禁止か判断するフラグ。デフォルト(false)では分割可能。
schemes	許可したいプロトコル	×	許可する URL スキーマ(プロトコル指定)をカンマ区切りで指定する。 デフォルトは http, https, ftp。

➢ バリデーション設定ファイル（validation.xml）の例

```
……
<form name="validate">
  <field property="urlField" depends="url">
    <arg key="urlField" position="0"/>
    <var>
      <var-name>allowallschemes</var-name>
      <var-value>false</var-value>
    </var>
    <var>
      <var-name>allow2slashes</var-name>
      <var-value>true</var-value>
    </var>
    <var>
      <var-name>nofragments</var-name>
      <var-value>false</var-value>
    </var>
    <var>
      <var-name>schemes</var-name>
      <var-value>http,ftp</var-value>
    </var>
  </field>
</form>
……
```

デフォルトのメッセージは「errors.url={0}には適切な URL 形式で入力してください.」

入力された値が URL の書式として正しいかをチェックする。

- arrayRange
  指定された属性の配列・コレクションの長さが、 指定数の範囲内であることをチェックする。
  - バリデーション設定ファイル（validation.xml）に設定を要する<var>要素

var-name	var-value	必須	概要
minArrayLength	最小配列数	×	配列の最小配列数を指定する。 最小配列数の指定がない場合、0が指定される。
maxArrayLength	最大配列数	×	配列の最大配列数を指定する。 最大配列数の指定がない場合、 int 型の最大値が指定される。

  - バリデーション設定ファイルサンプル（validation.xml）

```
……
<form name="validate">
  <field property="arrayField" depends="arrayRange">
    <arg key="arrayField" position="0"/>
    <var>
      <var-name>minArrayLength</var-name>
      <var-value>4</var-value>
    </var>
    <var>
      <var-name>maxArrayLength</var-name>
      <var-value>7</var-value>
    </var>
  </field>
</form>
……
```

デフォルトのメッセージは「errors.arrayRange={0}の要素数は{1}から{2}の範囲にしてください.」

入力された配列数が 4 個以上、7 個以下の範囲内であることをチェックする。

◆ 注意事項

- 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 TERASOLUNA Server Framework for Java 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。

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

◆ 構成クラス

	クラス名	概要
1	FieldChecks	半角チェック、数値チェックなど各種チェックロジックを定義した検証ルールクラス。
2	IndexedBeanWrapper	JavaBean の配列・コレクション型属性へのアクセサを持つインタフェース
3	IndexedBeanWrapperImpl	IndexedBeanWrapper インタフェースクラスの実装クラス。JavaBean の配列・コレクション型属性にアクセスできる。
4	BaseMultiFieldValidator	Validartor による相関チェックを行う場合に使用する抽象クラス。
5	TerasolunaController	サービス層のクラスを実行するリクエストコントローラ抽象クラス。本クラスにバリデータを設定する。
6	BindException	データバインドクラス及び形式チェッククラスから返却されるエラー情報を格納するクラス。 バインド対象及び形式チェック対象となる JavaBean を保持する。
7	ValidatorFactory	Validator を生成するクラスが実装すべきインタフェース
8	DefaultValidatorFactory	ValidatorFactory インタフェースクラスの実装クラス。オブジェクトの入力チェックを行う。
9	Validator	入力チェックを行うクラスが実装すべきインタフェース

◆ 拡張ポイント

なし

■ 関連機能

- 『RA-02 コントローラ拡張機能』
- 『RB-01 リクエストデータ解析機能』
- 『RD-01 例外ハンドリング機能』

■ 使用例

- TERASOLUNA Server Framework for Java (Rich 版) チュートリアル
  - 「2.5.2 単項目入力チェック」
  - 「2.5.3 相関入力チェック」
  - /webapps/WEB-INF/applicationContext.xml
  - /webapps/WEB-INF/tutorial-controller.xml
  - /webapps/WEB-INF/validation/*
  - jp.terasoluna.rich.tutorial.service.validation.AgeValidator.java 等

■ 備考

- その他の入力チェック
  - JavaBean1 クラス・2 パターンの入力チェックの設定
    チェック対象の JavaBean に対して、リクエストが異なる場合、異なる内容のチェックを行う。
    - ✧ サンプル
      ValidateSampleController の引数 TestBean に対して、SampleValidate1 からのリクエストと SampleValidate2 からのリクエストで異なる入力チェックを行う場合のサンプルを以下に記す。

- コントローラへの入力チェック機能の設定
  - Bean 定義サンプル

```
<!-- サンプル入力チェック業務 -->
……
<bean name="/SampleValidate1"
    class="jp.terasoluna.sample2.web.controller.ValidateSampleController"
    parent="xmlRequestController">
  <property name="sumService" ref="sumService"/>
  <property name="validator" ref="validator1"/>
</bean>
……
<bean name="/SampleValidate2"
    class="jp.terasoluna.sample2.web.controller.ValidateSampleController"
    parent="xmlRequestController">
  <property name="sumService" ref="sumService"/>
  <property name="validator" ref="validator2"/>
</bean>
……
```

- チェックルールの定義
  - Bean 定義サンプル

```
<!-- バリデータ1 -->
<bean id="validator1"
    class="org.springmodules.commons.validator.ConfigurableBeanValidator">
  <property name="validatorFactory" ref="validatorFactory"/>
  <property name="formName" value="testBeanA"/>
</bean>

<!-- バリデータ2 -->
<bean id="validator2"
    class="org.springmodules.commons.validator.ConfigurableBeanValidator">
  <property name="validatorFactory" ref="validatorFactory"/>
  <property name="formName" value="testBeanB"/>
</bean>
```

バリデータ名を指定する。

form 名を指定する。下記、『チェック対象属性とチェックルールの関連付け』で設定した form 名を指定する。

- チェック対象属性とチェックルールの関連付け
  - バリデーション定義ファイルサンプル(validation.xml)

```
……
<!-- 5文字までの文字列チェック -->
  <form name="testBeanA">
    <field property="userName" depend="stringLength"/>
  </form>
……
<!-- 10文字までの文字列チェック -->
  <form name="testBeanB">
    <field property="userName" depend="stringLength"/>
  </form>
……
```

form 名を指定する。
検査対象の JavaBean クラス名である必要はない。

    - このパターンを利用する場合、『コーディングによる入力チェック』パターンは利用出来ない。

◆ 使用例

- TERASOLUNA Server Framework for Java (Rich 版) 機能網羅サンプル
  - UC109 入力チェック
    - jp.terasoluna.rich.functionsample.inputcheck.*