NTT DATA

TERASOLUNA Server Framework for .NET

機能説明書

第 2.1.0.1 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 .NET 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 は、Apache Software Foundation の登録商標または商標です。
・Java 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。
・Microsoft、Visual Stuido、Windows、.NET Framework 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。
・TERASOLUNA は、株式会社 NTT データの登録商標です。
・その他の会社名、製品名は、各社の登録商標または商標です。

本書は、以下のフレームワークに対応しています。

- TERASOLUNA Server Framework for .NET ver2.1 系

目次

CM-01 メッセージ管理機能

■ 概要

本機能では、アプリケーションで扱うメッセージに対し、統一的にアクセスする仕組みを提供する。

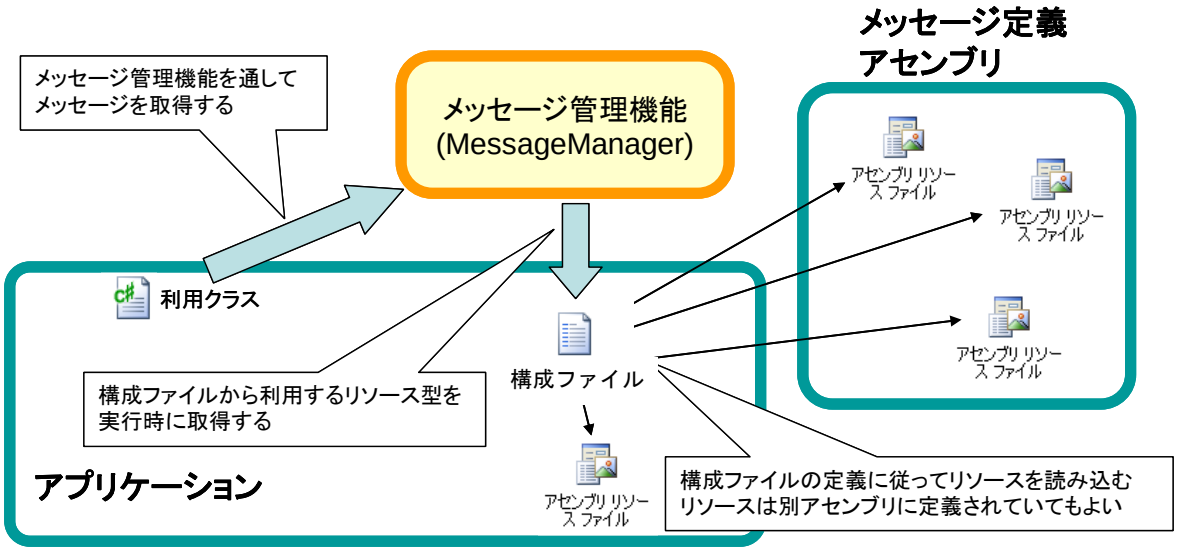


図 1 動作概念図

メッセージ管理機能を利用してメッセージを取得するには、MessageManager クラスのメソッドを呼び出す。MessageManager は、構成ファイル[1]の定義に従ってアセンブリリソースファイルを読み込み、メッセージを返却する。メッセージを取得するクラスと、アセンブリリソースファイルは、異なるアセンブリでもよい。

[1] 構成ファイル：web.config や App.config のことを示す。

■ 使用方法

◆ 構成ファイル

本機能を有効にする場合、構成ファイルに設定を定義する。

表 1 構成ファイル

ノード	属性	必須	値
/configuration/appSettings/add			複数可
	key	○	一意の文字列 MessageResources.MyMessage01
	value	○	アセンブリ修飾名

● メッセージリソースの指定

利用するメッセージリソースの指定は構成ファイルの appSettings セクションに、”MessageResource.”をプレフィックスとする key 属性を持つ add 要素を追加することで行う。add 要素の value 属性には、利用するメッセージリソースの型をアセンブリ修飾名で記述する。メッセージリソースの型は、メッセージを利用するクラスと同一のアセンブリである必要はない。異なるアセンブリのメッセージを利用する場合には、実行時に読み込むことが可能な場所にアセンブリを配置する。

以下に構成ファイルの設定例を示す。

```
<appSettings>
  <add key="MessageResources.MyMessage00"
       value="CommonResources.CommonResource, CommonResources"/>
  <add key="MessageResources.MyMessage01"
       value="MessageResources.Resource01, MessageResources"/>
  <add key="MessageResources.MyMessage02"
       value="MessageResources.Resource02, MessageResources"/>
</appSettings>
```

リスト 1 構成ファイルの例

● 複数リソースの指定

リスト 1 の例では、”MessageResources.”をプレフィックスとする key を持つ add 要素が 3 つ記述されている。MessageManager は、構成ファイルに”MessageResources.”をプレフィックスとする key を持つ add 要素が複数定義されている場合には、それらをすべて読み込み、メッセージの検索対象とする。その際、複数のリソース間のメッセージ検索順は、key 属性に指定された文字列をキーとした辞書順となることに注意する。リスト 1 の例では、まず CommonResource のメッセージが検索され、CommonResource に指定されたメッセージ ID が存在していない場合のみ Resource01 のメッセージが検索される。

◆ 実装方法

● メッセージリソースの作成

本機能において利用できるリソースは、.NET Framework が提供する ResourceManager クラスが対応するリソース型及びリソースファイル[2]に準ずる。メッセージを格納するリソース型を作成する場合、Visual Studio のプルダウンメニューより、[プロジェクト - 新しい項目の追加]を選択し、「新しい項目の追加」ダイアログのテンプレートの一覧から「アセンブリ リソースファイル」を追加する。

以下に、Visual Studio からプロジェクトにリソースを追加するイメージを示す。

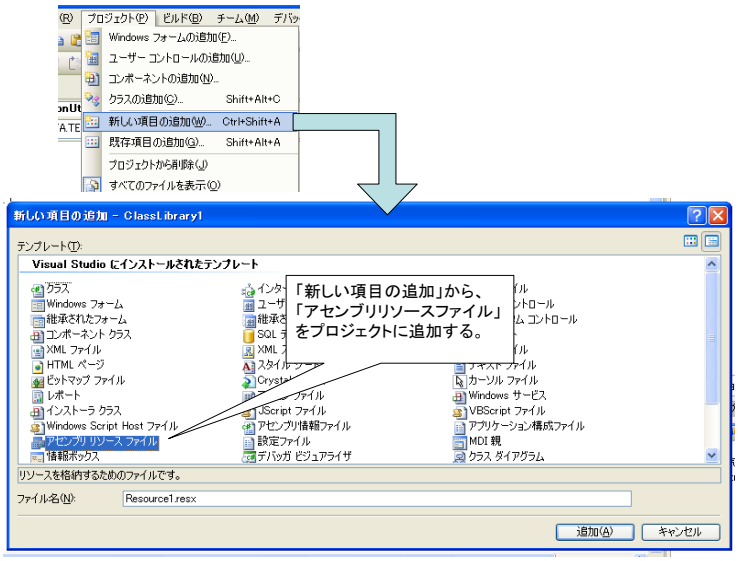


図 2 アセンブリリソースファイルの追加

リソースを作成するプロジェクトは、メッセージリソースを利用するクラスと同じアセンブリである必要はない。また、全てのメッセージリソースが一つのアセンブリに格納される必要はなく、コンパイル時に参照が解決される必要もない。実行時にはアプリケーションから参照可能な場所にメッセージリソースが定義されたアセンブリを配置する。

● メッセージリソースの編集

作成したリソース型をソリューションエクスプローラでダブルクリックするか、「ファイルを開くアプリケーションの選択」から「マネージリソースエディタ」を選択することで、リソースエディタが表示される。リソースエディタでリソースを編集すると、自動的に「[リソース名].Designers.cs」という名称でソースコードが自動生成され、リソースにアクセスするための型が作成される。リソースにアクセスするための型は、リソース名が型名となる。メッセージ管理機能では、この型を指定することで、メッセージを取得してくるリソースを追加することができる。

以下に、メッセージリソースの編集イメージを示す。

---

[2] リソースについては MSDN ドキュメント「.NET Framework 開発者ガイド - リソースのパッケージ化と配置」を参照のこと。

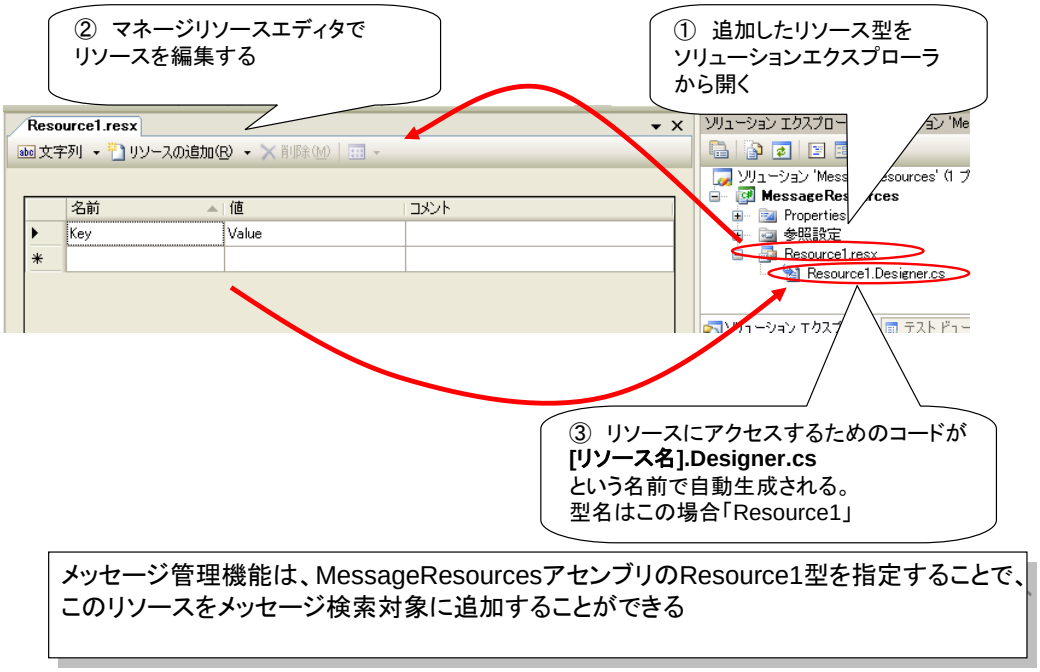


図 3 Visual Studio によるメッセージリソースの編集イメージ

なお、マネージリソースエディタでは、文字列リソースの他、アイコンや画像、ファイルなどもリソースに埋め込むことができるが、本機能によって利用できるリソースは文字列リソースのみとなり、イメージ、アイコン、オーディオ、ファイルなどには対応しない。リソースを指定する場合には、自動生成されたリソース型をアセンブリ修飾名[3]で指定する。

- メッセージの利用

メッセージ管理機能を用いてメッセージを取得する場合には、まずプロジェクトの参照設定に TERASOLUNA.Fw.Common アセンブリを追加する。ソースコードのメッセージを利用する場所では、以下のようにしてメッセージを取得するためのメッセージ ID を用いてメッセージを取得する。

```
string myMessage = MessageManager.GetMessage("MessageKey");
```

リスト 2 メッセージ管理機能の利用の例

MessageManager クラスは静的なメソッド GetMessage(string ,params object[])と静的なプロパティ Culture を公開する。

MessageManager は、GetMessage メソッドの第一引数として渡したメッセージ ID を用いて、管理するリソースの中にメッセージ ID に対応するメッセージが定義されているか検索する。存在すればその文字列を返し、存在しなければ null 参照を返す。複数のリソースで同一のメッセージ

[3] アセンブリ修飾名については MSDN ドキュメント「.NET Framework 開発者ガイド - 完全修飾型名の指定」を参照のこと。

ID に合致するメッセージが定義されている場合、最初に見つかったメッセージを返す。複数のリソースが存在する場合の検索順に関しては MessageManager が初期化される段階で決定される。

Culture プロパティを用いて、検索するメッセージのカルチャを指定できる。日本語のメッセージの他、利用される環境の言語環境に応じたメッセージを用意することが可能である。

- 書式項目の利用

MessageManager の GetMessage メソッドには、可変長引数として、複数のオブジェクト args を渡すことができる。args に一つ以上のオブジェクトが指定された場合、GetMessage メソッドはリソースから取得したメッセージを args で指定されたオブジェクトを用いてフォーマットする。このとき、メッセージには与えられた args のオブジェクトに対応したインデックス付プレースホルダ(書式項目)が適切な数定義されていなければならない。

以下に、メッセージを書式設定する例を示す。リスト 5 はリソースに定義されているメッセージであるとする。

```
ARG_NULL_EXCEPTION = 引数 "{0}" が null 参照です.
```

リスト 3 書式項目を利用したメッセージ定義例

メッセージ利用側では、たとえば書式項目{0}に対応する文字列"user_name"を GetMessage メソッドの第二引数に与えることで、「引数 "user_name" が null 参照です.」というメッセージを取得することができる。

```
string myMessage =
            MessageManager.GetMessage("ARG_NULL_EXCEPTION", "user_name");
```

リスト 4 置換文字列の利用例

置換文字列を利用したメッセージの整形については、.NET Framework の提供する複合書式設定機能[4]を参照のこと。なお、書式項目を利用したフォーマットを行う場合、args に指定したオブジェクトの数以上のインデックスを持つプレースホルダがメッセージに存在すると例外が発生するため、注意すること。

たとえば、以下のような定義がされている場合に、GetMessage メソッドの args として 4 つ以上のオブジェクトを指定しない場合、例外となる。

```
ARG_EXCEPTION = 引数 "{3}" は不正です.   /* args が 3 つ以下だと例外! */
```

リスト 5 危険なメッセージ定義例

◆ 構成クラス

表 2 構成クラス一覧

[4] 複合書式設定については MSDN ドキュメント「.NET Framework 開発者ガイド -複合書式設定」を参照のこと。

項番	クラス名	説明
1	MessageManager	メッセージ管理機能を提供するユーティリティクラス

■ 拡張ポイント

独自のメッセージ管理クラスをフレームワークに沿って利用する場合、MessageManager を継承したクラスを作成する。その際、Init メソッドをオーバーライドして ResourceManagerList の初期化ロジックを変更することで、利用するメッセージリソースの読み込み方法や、検索順序をカスタマイズすることができる。

- 利用するメッセージ管理クラス実装の変更

メッセージ管理機能を提供する MessageManager は内部に MessageManager クラスまたは MessageManager 派生クラスのインスタンスを一度だけ生成して利用する。このとき、構成ファイルの appSettings セクションにメッセージ管理クラスの型のアセンブリ修飾名を指定することで、利用する実装を差し替えることができる。キーは”MessageManagerTypeName” とする。指定を省略した場合、MessageManager クラスのインスタンスが利用される。

以下に、メッセージ管理クラスの差し替えを設定する例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="MessageManagerTypeName"
         value="MyNamespace.MessageManagerEx, MyNamespace"/>
  </appSettings>
</configuration>
```

リスト 6 メッセージ管理クラス指定の例

■ 関連機能

特になし

CM-02 入力値検証機能

■ 概要

本機能は、入力値検証設定ファイルに定義した検証ルールに従って、入力値検証を行う機能を提供する。検証対象は、データセットが保持するテーブルの各カラムであり、カラムに格納されている値が検証ルールと一致するかをチェックする。必須入力チェック、半角カナ文字列チェック、日付形式チェック、int 型範囲チェックなど、業務アプリケーションで必要な各種検証ルールを提供している。詳細は後述の『入力値検証ルール解説』を参照のこと。

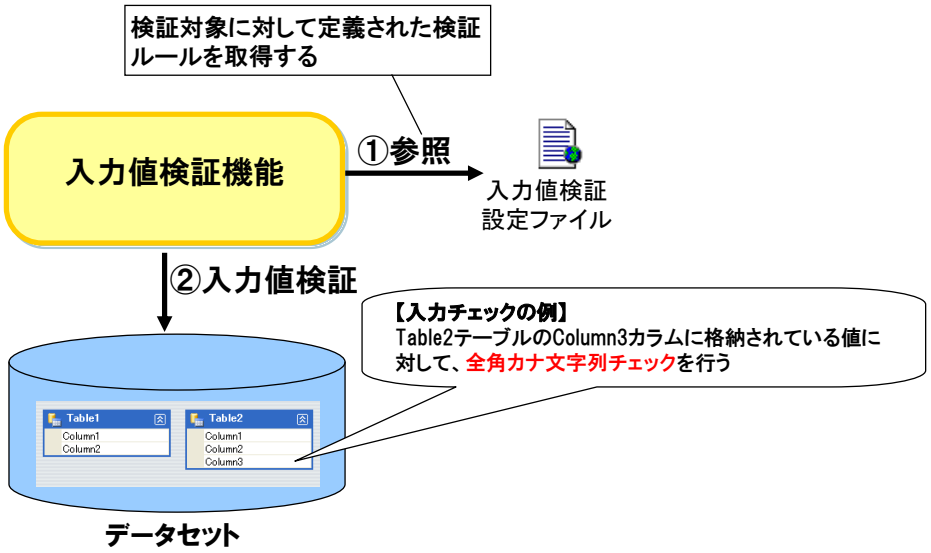


図 1 動作概念図

入力値検証機能は、入力値検証設定ファイルを参照し、検証対象（データセットが保持するテーブルの各カラム）に対して定義された検証ルールを取得する。そして、取得した検証ルールに従って、検証対象に対して入力値検証を行う。

データセットは複数のテーブルを持ち、また、各テーブルには複数のカラムが存在する。そのため、入力値検証設定ファイルには、「どのデータセットの、どのテーブルの、どのカラムに対して、どの検証ルールを適用するか」を定義する。

1 つのカラムに対して、複数の検証ルールを適用することが可能である。例えば、あるカラムの入力値が必須かつ全角文字列でなければならない場合、「必須入力チェック」と「全角文字列チェック」を組み合わせることで実現できる。

なお、本機能は主にイベント処理機能とリクエストコントローラ機能から呼び出される。

■ 使用方法

◆ 入力値検証設定ファイル

検証対象に対してどの検証ルールを適用するかを、入力値検証設定ファイルに定義する。

表 1 入力値検証設定ファイル

ノード	属性	必須	値
/configuration/configSections/section			複数可
	name	○	構成要素名。 固定値、以下を指定する。 validation
	type	○	構成設定の処理を行う構成セクションハンドラクラス名。 固定値、以下を指定する。 Microsoft.Practices.EnterpriseLibrary.Validation.Configuration.ValidationSettings, Microsoft.Practices.EnterpriseLibrary.Validation, Version=5.0.414.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35
/configuration/validation/type			複数可
	assenblyName	○	検証対象のアセンブリ名。
	name	○	検証対象のデータ行クラスの型名。 データセット内のテーブルは、データセットクラスの内部クラスとして定義されており、このクラスのことを「データ行クラス」という。 データ行クラスの型名は、データセットクラス名とデータ行クラス名を"+"で連結して次のように記述する。 "データセットクラス名＋テーブル行クラス名"
/configuration/validation/type/ruleset			複数可
	name	○	ルールセット名。 /configuration/validation/type 要素で指定したテーブルに対して、検証設定をルールセットという単位で複数定義することができる。ルールセット名は ruleset 要素の中で一意でなければならない。 イベント処理機能やリクエストコントローラ機能では、ルールセット名の既定値と

			して”Default”を利用する。したがって、それらの機能でルールセット名の設定を省略して既定値を利用する場合は、入力値検証設定ファイルの ruleset 要素 name 属性に”Default”と設定すること。
/configuration/validation/ruleset/properties/property			複数可
	name	○	検証対象カラム名。 同一テーブルの複数のカラムに対して入力値検証を行う場合は、property 要素をカラムの数だけ追加する。
/configuration/validation/ruleset/properties/property/validator			複数可
	適用する検証ルールに応じて、必要な属性を設定する。 複数の検証ルールを組み合わせて適用したい場合、validator 要素を検証ルールの数だけ追加する。		

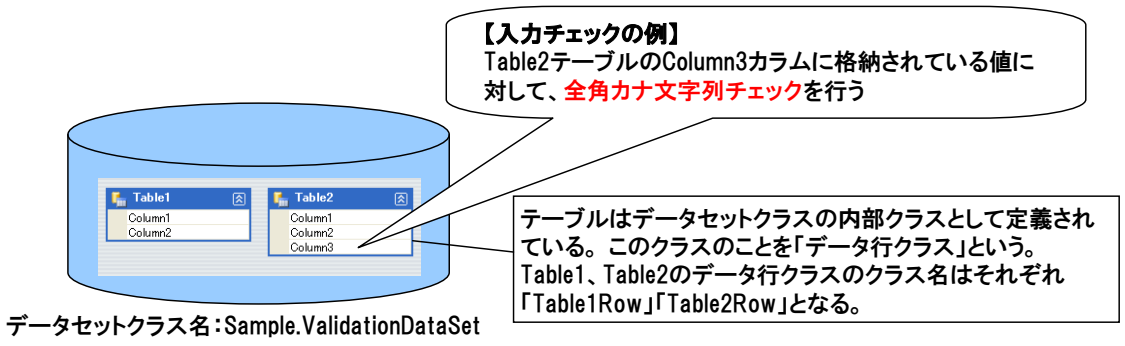


図 2 入力チェックの例

図 2 に示すような入力チェック(Table2 テーブルの Column3 カラムに格納されている値に対する全角カナ文字列チェック)を行う場合の、入力値検証設定ファイルの設定例を以下に示す。

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="validation"
      type="Microsoft.Practices.EnterpriseLibrary.Validation.Configuration.ValidationSettings, Microsoft.Practices.EnterpriseLibrary.Validation, Version=5.0.414.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" />
  </configSections>
  <validation>
    <type assemblyName="Test"
          name="Sample.ValidationDataSet+Table2Row">
      <ruleset name="customRuleSet">
        <properties>
          <property name="Column3">
            <validator negated="false"
                       tag="Column1"
                       name="ZenkakuKanaStringValidator"
                       type="TERASOLUNA.Fw.Common.Validation.Validators.ZenkakuKanaStringValidator, TERASOLUNA.Fw.Common"/>
          </property>
        </properties>
      </ruleset>
    </type>
  </validation>
</configuration>
```

データセットクラス名＋データ行クラス名

ルールセット名

カラム名

リスト 1 入力値検証設定ファイルの設定例

validator 要素には、適用する検証ルールに応じて、必要な属性を設定する。各検証ルールに共通の属性を以下に示す。

表 2 validator 要素に設定する検証ルール共通の属性

項番	属性名	必須	説明	デフォルト値
1	negated		true を設定すると検証ルールを反転してチェックする。true もしくは false を設定する。	false
2	messageTemplate		検証エラーメッセージのテンプレートとして使用する文字列。	各検証ルールに定義されている既定のメッセージ
3	messageTemplateResourceName		検証エラーメッセージのテンプレートをメッセージリソースから読み込む際の名前。	なし
4	messageTemplateResourceType		検証エラーメッセージのテンプレートをメッセージリソースから読み込む際のリソースの型。	なし
5	tag		メッセージテンプレートのプレースホルダ{2}に渡される文字列。	なし

			検証対象項目の論理名を設定する。	
6	type	○	適用する検証ルールに対応した検証クラスの型を、完全修飾名で設定する。	なし
7	name	○	検証ルールの名前。任意の名前を設定することができる。	なし

共通の属性以外に、検証ルール固有の属性が存在する。例えば範囲チェックを行うルールならば上限値と下限値を指定するための属性がある。固有の属性の詳細については、後述の『入力値検証ルール解説』を参照のこと。

- 検証エラーメッセージのテンプレートの設定

  messageTemplate 属性に設定した文字列、またはメッセージリソースから読み込んだ文字列は、検証エラーが発生した際に検証エラーメッセージを生成するのに用いられるテンプレートである。プレースホルダ{0}～{2}には固定で以下の値が格納される。

表 3 メッセージテンプレートに渡される文字列

プレースホルダ	説明
{0}	検証対象のカラムに格納されている値の文字列表現
{1}	カラム名(検証対象のカラム名)
{2}	validator 要素の tag 属性に設定した文字列

{3}以降のプレースホルダについては、検証ルールによって利用形態が異なる。詳細については後述の『入力値検証ルール解説』を参照のこと。

- 適用する検証ルールに対応した検証クラスの設定

  type 属性には、適用する検証ルールに対応した検証クラスの型をアセンブリ修飾名で設定する。検証ルールは次の 3 つの提供元がある。

  - Validation Application Block[1]
  - Validation Application Block Extensions[2]
  - TERASOLUNA フレームワーク[3]

---

[1] Validation AB が提供する検証クラスの名前空間は
“Microsoft.Practices.EnterpriseLibrary.Validation.Validators”となり、アセンブリ名は
“Microsoft.Practices.EnterpriseLibrary.Validation” となる。

[2] Validation Application Block Extensions が提供する検証クラスの名前空間は、
“EntLibContrib.Validation.Validators”となり、アセンブリは”EntLibContrib.Validation”となる。

[3] TERASOLUNA フレームワークが提供する検証クラスの名前空間は、
“TERASOLUNA.Fw.Common.Validation.Validators”となり、アセンブリ名は
“TERASOLUNA.Fw.Common”となる。

以下に、各提供元が提供する検証ルールとその検証クラスを示す。

表 4 Validation Application Block が提供する検証ルール

項番	検証ルール	検証クラス	概要
1	byte 型チェック	TypeConversionValidator	検証対象が byte(Byte)型に変換可能か検証する。
2	short 型チェック		検証対象が short(Int16)型に変換可能か検証する。
3	int 型チェック		検証対象が int(Int32)型に変換可能か検証する。
4	long 型チェック		検証対象が long(Int64)型に変換可能か検証する。
5	decimal 型チェック		検証対象が decimal(Decimal)型に変換可能か検証する。
6	float 型チェック		検証対象が float(Single)型に変換可能か検証する。
7	double 型チェック		検証対象が double(Double)型に変換可能か検証する。

表 5 Validation Application Block Extensions が提供する検証ルール

項番	検証ルール	検証クラス	概要
1	要素数チェック	CollectionCountValidator	検証対象がコレクション(List や配列など)であった場合、指定した要素数の範囲であるかを検証する。

表 6 TERASOLUNA フレームワークが提供する検証ルール

項番	検証ルール	検証クラス	概要
1	正規表現一致チェック	RegexValidatorEx	正規表現パターンを指定して、検証対象文字列がパターンとマッチするかを検証する。
2	必須入力チェック	RequiredValidator	検証対象が null またはホワイトスペース(半角空白、全角空白、改行、タブ文字)でないか検証する。
3	数値文字列チェック	NumericStringValidator	検証対象が半角数値文字のみで構成されているか検証する。
4	半角英数大文字列チェック	CapAlphaNumericStringValidator	検証対象が大文字の半角英数文字のみで構成されているか検証する。

5	半角英数文字列チェック	AlphaNumericStringValidator	検証対象が半角英数文字のみで構成されているか検証する。
6	半角文字列チェック	HankakuStringValidator	検証対象が半角文字のみで構成されているか検証する。
7	半角カナ文字列チェック	HankakuKanaStringValidator	検証対象が半角カナ文字のみで構成されているか検証する。
8	全角カナ文字列チェック	ZenkakuKanaStringValidator	検証対象が全角カナ文字のみで構成されているか検証する。
9	全角文字列チェック	ZenkakuStringValidator	検証対象が全角文字のみで構成されているか検証する。
10	URL 形式チェック	UrlValidator	検証対象が URL 形式の文字列であるか検証する。
11	日付形式チェック	DateTimeFormatValidator	検証対象が日付形式の文字列であるか検証する。
12	int 型範囲チェック	IntRangeValidator	検証対象が int 型に変換可能であり、指定した範囲であるか検証する。
13	decimal 型範囲チェック	DecimalRangeValidator	検証対象が decimal 型に変換可能であり、指定した範囲であるか検証する。
14	float 型範囲チェック	FloatRangeValidator	検証対象が float 型に変換可能であり、指定した範囲であるか検証する。
15	double 型範囲チェック	DoubleRangeValidator	検証対象が double 型に変換可能であり、指定した範囲であるか検証する。
16	日付型範囲チェック	DateTimeRangeValidatorEx	検証対象が dateTime 型に変換可能であり、指定した範囲の日時であるかを検証する。
17	byte 列長範囲チェック	ByteRangeValidator	検証対象の文字列を指定したエンコーディングでバイト列に展開した際のバイト長が指定した範囲であるか検証する。
18	文字列長チェック	StringLengthRangeValidator	文字列が指定した文字数の範囲であるかを検証する。
19	必須文字列チェック	ContainsCharactersValidatorEx	指定した文字列を含んでいるかどうかを検証する。
20	数値チェック	NumberValidator	検証対象が数値に変換可能であり、整数部・小数部がそれぞれ指定した桁数であるか検証する。

◆ 実装方法

本機能は、主に「イベント処理機能」と「リクエストコントローラ機能」から呼び出される。本機能を利用する際に必要な共通設定についての説明と、各機能からの呼び出し例について説明する。

- 本機能を利用する際に必要な共通設定

  本機能を利用するには、プロジェクトの参照設定に次の dll を追加する。

  - EntLibContrib.Validation
  - Microsoft.Practices.EnterpriseLibrary.Common
  - Microsoft.Practices.EnterpriseLibrary.Validation
  - TERASOLUNA.Fw.Common

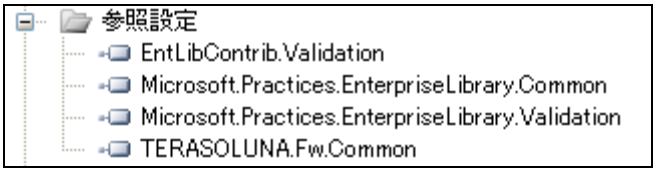


図 3 参照設定に追加した dll

- イベント処理機能からの呼び出し例

  EventController コンポーネントの ValidationFilePath プロパティに入力値検証設定ファイルのパスを指定することで、入力値検証機能を利用できる。イベント処理機能から呼び出す際の実装の流れを以下に示す。

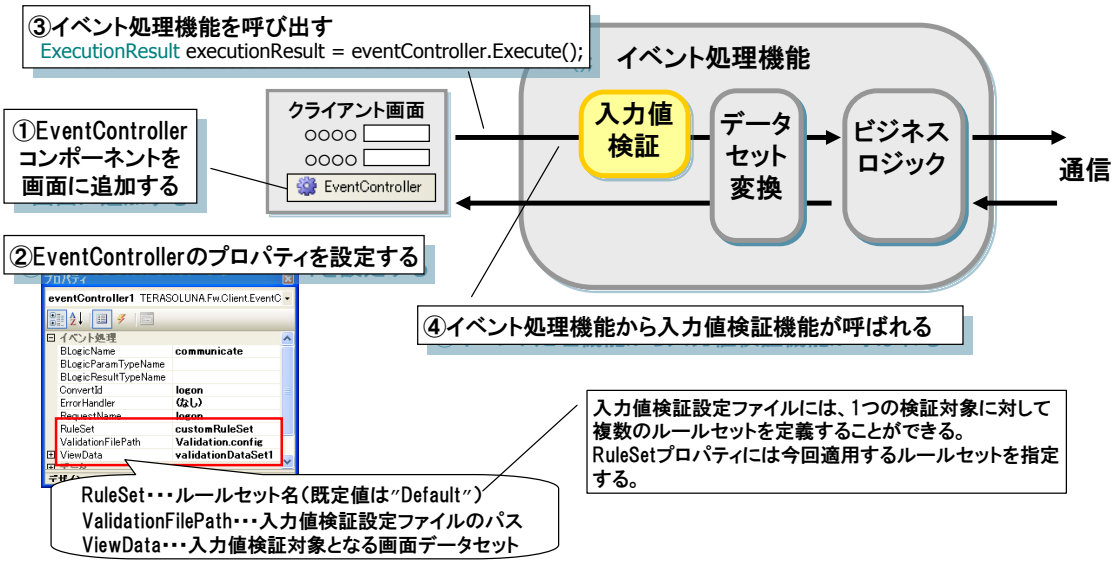


図 4 イベント処理機能から呼び出す際の実装の流れ

- リクエストコントローラ機能からの呼び出し例

  ビジネスロジッククラスのクラス属性に、 ValidationFilePath を定義することで、ビジネスロジックの入力値に対する検証を行うことができる。リクエストコントローラ機能から呼び出す際の実装例を以下に示す。

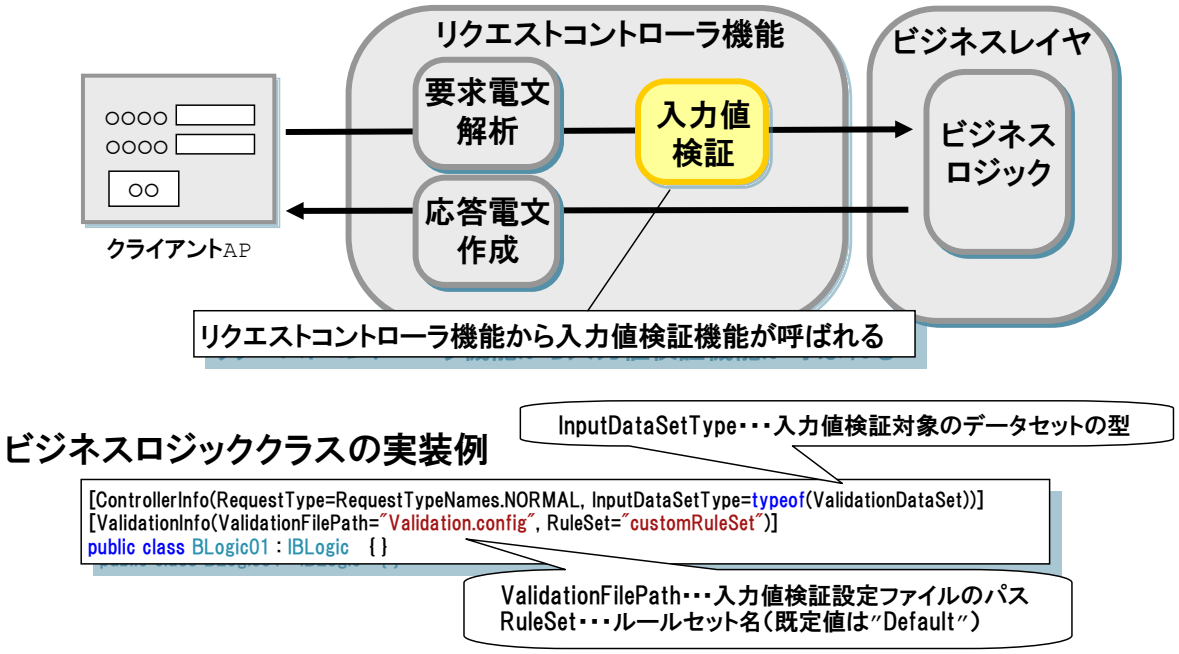


図 5 リクエストコントローラ機能から呼び出す際の実装例

- 各機能から呼び出す際の設定のイメージ

  イベント処理機能／リクエストコントローラ機能から呼び出す際の設定と、入力値検証設定ファイル／入力値検証対象のデータセットとの関係のイメージを以下に示す。

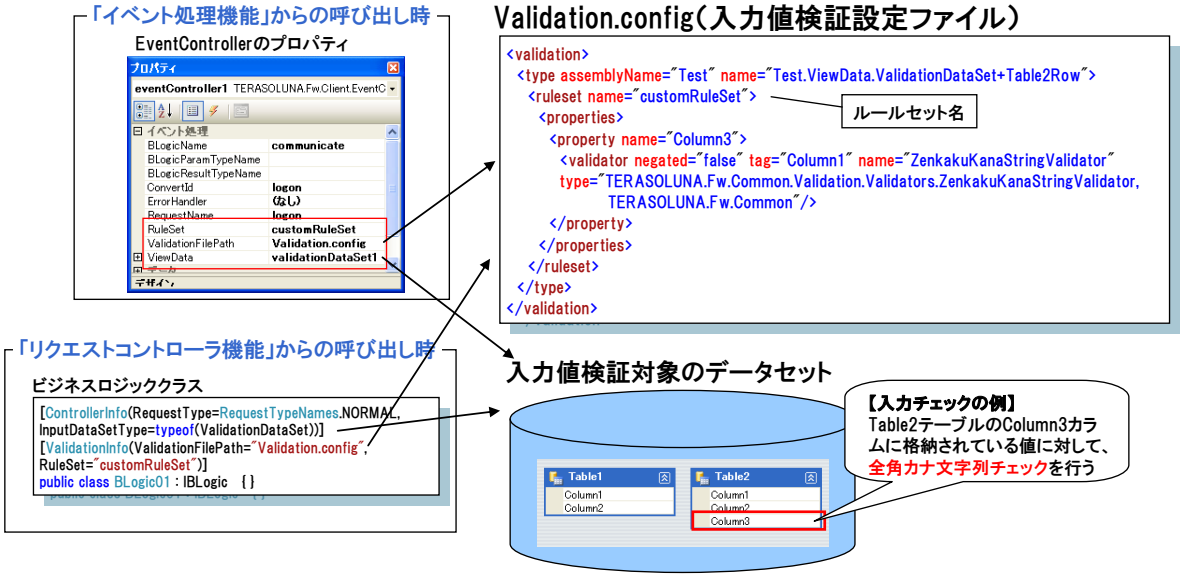


図 6 各機能から呼び出す際の設定のイメージ

■ 入力値検証ルール解説

◆ Validation Application Block が提供する検証ルール解説

Validation Application Block(以下、Validation AB)が提供する入力値検証ルールの機能と利用方法について解説する。

Validation AB が提供する検証クラスの名前空間は“Microsoft.Practices.EnterpriseLibrary.Validation.Validators”となり、アセンブリ名は“Microsoft.Practices.EnterpriseLibrary.Validation” となる。

- byte 型チェック
  検証対象値が byte 値(System.Byte)に変換可能かを検証する。Validation AB が提供する TypeConversionValidator を利用する。

表 7 構成クラス

項番	クラス名	説明
1	TypeConversionValidator	指定した型に変換可能かを検証する検証クラス
2	TypeConversionValidatorData	設定情報から TypeConversionValidator を生成、初期化するクラス

表 8 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	targetType	○	チェック対象の型を指定する。byte 型チェックでは、System.Byte を指定する	なし

表 9 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	変換対象の型名(System.Byte)

以下に設定ファイルへの設定例を示す。

```
<validator targetType="System.Byte"
  negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.TypeConversionValidator,
  Microsoft.Practices.EnterpriseLibrary.Validation, Version=5.0.414.0, Culture=neutral,
  PublicKeyToken=31bf3856ad364e35"
  name="Byte Type Conversion Validator" />
```

リスト 2 設定例

- short 型チェック
  検証対象値が short 値(System.Int16)に変換可能かを検証する。Validation AB が提供する TypeConversionValidator を利用する。

表 10 構成クラス

項番	クラス名	説明
1	TypeConversionValidator	指定した型に変換可能かを検証する検証クラス
2	TypeConversionValidatorData	設定情報から TypeConversionValidator を生成、初期化するクラス

表 11 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	targetType	○	チェック対象の型を指定する。short 型チェックでは、System.Int16 を指定する	なし

表 12 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	変換対象の型名(System.Int16)

以下に設定ファイルへの設定例を示す。

```
<validator targetType="System.Int16"
  negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.TypeConversionValidator,
  Microsoft.Practices.EnterpriseLibrary.Validation, Version=5.0.414.0, Culture=neutral,
  PublicKeyToken=31bf3856ad364e35"
  name="Short Type Conversion Validator" />
```

リスト 3 設定例

- int 型チェック
  検証対象値が int 値(System.Int32)に変換可能かを検証する。Validation AB が提供する TypeConversionValidator を利用する。

表 13 構成クラス

項番	クラス名	説明
1	TypeConversionValidator	指定した型に変換可能かを検証する検証クラス
2	TypeConversionValidatorData	設定情報から TypeConversionValidator を生成、初期化するクラス

表 14 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	targetType	○	チェック対象の型を指定する。int 型チェックでは、System.Int32 を指定する	なし

表 15 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	変換対象の型名(System.Int32)

以下に設定ファイルへの設定例を示す。

```
<validator targetType="System.Int32"
  negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.TypeConversionValidator,
  Microsoft.Practices.EnterpriseLibrary.Validation, Version=5.0.414.0, Culture=neutral,
  PublicKeyToken=31bf3856ad364e35"
  name="Int Type Conversion Validator" />
```

リスト 4 設定例

- long 型チェック
  検証対象値が long 値(System.Int64)に変換可能かを検証する。Validation AB が提供する TypeConversionValidator を利用する。

表 16 構成クラス

項番	クラス名	説明
1	TypeConversionValidator	指定した型に変換可能かを検証する検証クラス
2	TypeConversionValidatorData	設定情報から TypeConversionValidator を生成、初期化するクラス

表 17 固有設定項目

項番	属性名	必須	説明	設定値
1	targetType	○	チェック対象の型を指定する。long 型チェックでは、System.Int64 を指定する	なし

表 18 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	変換対象の型名(System.Int64)

以下に設定ファイルへの設定例を示す。

```
<validator targetType="System.Int64"
  negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.TypeConversionValidator,
  Microsoft.Practices.EnterpriseLibrary.Validation, Version=5.0.414.0, Culture=neutral,
  PublicKeyToken=31bf3856ad364e35"
  name="Long Type Conversion Validator" />
```

リスト 5 設定例

- decimal 型チェック
  検証対象値が decimal 値(System.Decimal)に変換可能かを検証する。Validation AB が提供する TypeConversionValidator を利用する。

表 19 構成クラス

項番	クラス名	説明
1	TypeConversionValidator	指定した型に変換可能かを検証する検証クラス
2	TypeConversionValidatorData	設定情報から TypeConversionValidator を生成、初期化するクラス

表 20 固有設定項目

項番	属性名	必須	説明	設定値
1	targetType	○	チェック対象の型を指定する。decimal 型チェックでは、System.Decimal を指定する	なし

表 21 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	変換対象の型名(System.Decimal)

以下に設定ファイルへの設定例を示す。

```
<validator targetType="System.Decimal"
  negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.TypeConversionValidator,
  Microsoft.Practices.EnterpriseLibrary.Validation, Version=5.0.414.0, Culture=neutral,
  PublicKeyToken=31bf3856ad364e35"
  name="Decimal Type Conversion Validator" />
```

リスト 6 設定例

- float 型チェック
  検証対象値が float 値(System.Single)に変換可能かを検証する。Validation AB が提供する TypeConversionValidator を利用する。

表 22 構成クラス

項番	クラス名	説明
1	TypeConversionValidator	指定した型に変換可能かを検証する検証クラス
2	TypeConversionValidatorData	設定情報から TypeConversionValidator を生成、初期化するクラス

表 23 固有設定項目

項番	属性名	必須	説明	設定値
1	targetType	○	チェック対象の型を指定する。float 型チェックでは、System.Single を指定する	なし

表 24 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	変換対象の型名(System.Single)

以下に設定ファイルへの設定例を示す。

```
<validator targetType="System.Single"
  negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.TypeConversionValidator,
  Microsoft.Practices.EnterpriseLibrary.Validation, Version=5.0.414.0, Culture=neutral,
  PublicKeyToken=31bf3856ad364e35"
  name="Float Type Conversion Validator" />
```

リスト 7 設定例

- double 型チェック
  検証対象値が double 値(System.Double)に変換可能かを検証する。Validation AB が提供する TypeConversionValidator を利用する。

表 25 構成クラス

項番	クラス名	説明
1	TypeConversionValidator	指定した型に変換可能かを検証する検証クラス
2	TypeConversionValidatorData	設定情報から TypeConversionValidator を生成、初期化するクラス

表 26 固有設定項目

項番	属性名	必須	説明	設定値
1	targetType	○	チェック対象の型を指定する。double 型チェックでは、System.Double を指定する	なし

表 27 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	変換対象の型名(System.Double)

以下に設定ファイルへの設定例を示す。

```
<validator targetType="System.Double"
  negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="Microsoft.Practices.EnterpriseLibrary.Validation.Validators.TypeConversionValidator,
  Microsoft.Practices.EnterpriseLibrary.Validation, Version=5.0.414.0, Culture=neutral,
  PublicKeyToken=31bf3856ad364e35"
  name="Double Type Conversion Validator" />
```

リスト 8 設定例

◆ Validation Application Block Extensions が提供する入力値検証ルールの解説

Enterprise Library Contrib の Validation Application Block Extensions（以下、Validation AB Ex）が提供する入力値検証ルールを説明する。

Validation AB Ex が提供する検証クラスの名前空間は、“EntLibContrib.Validation.Validators”となり、アセンブリは”EntLibContrib.Validation”となる。

- 要素数チェック

  検証対象が配列や List などコレクション(ICollection)型であり、要素数が指定した範囲に含まれているか検証する。Validation AB Extensions で提供する CollectionCountValidator を利用する。なお、要素数は ICollection インターフェイスで定義された Count プロパティの値である。配列の場合は Length プロパティの値となる。

  文字列長チェックで示した範囲チェックについての注意点も参考のこと。

表 28 構成クラス

項番	クラス名	説明
1	CollectionCountValidator	要素数チェックを行う検証クラス
2	CollectionCountValidatorData	要素数チェックを行う CollectionCountValidator を生成するクラス

表 29 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	lowerBound	-	検証対象である要素数の範囲の最小値を指定する	0
2	lowerBoundType	-	lowerBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Ignore
3	upperBound	-	検証対象である要素数の最大値を指定する	0
4	upperBoundType	-	upperBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Inclusive

表 30 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	最小値(lowerBound)
2	{4}	最小値の値の扱い(lowerBoundType)
3	{5}	最大値(upperBound)
4	{6}	最大値の値の扱い(upperBoundType)

以下に設定ファイルへの設定例を示す。

```
<validator lowerBound="1" lowerBoundType="Exclusive" upperBound="256"
  upperBoundType="Exclusive" negated="false" messageTemplate=""
  messageTemplateResourceName="" messageTemplateResourceType="" tag=""
  type="EntLibContrib.Validation.Validators.CollectionCountValidator,
        EntLibContrib.Validation"
  name="Collection Count Validator" />
```

リスト 9 設定例

◆ フレームワークが提供する入力値検証ルールの解説

フレームワークが提供する入力値検証ルールの機能と利用方法について解説する。

フレームワークが提供する検証クラスの名前空間は
“TERASOLUNA.Fw.Common.Validation.Validators”となり、アセンブリ名は
“TERASOLUNA.Fw.Common”となる。

- 正規表現一致チェック
  検証対象値が指定した正規表現文字列にマッチするかどうかを検証する。正規表現パターンとともに、.NET Framework で提供される正規表現エンジンの各種オプションを利用することができる。フレームワークが提供する RegexValidatorEx を利用する。

  以下に、検証成功となるパターンを示す。

  - 通常時
    検証対象となる文字列が指定されたパターンと正規表現オプションによって表される正規表現にマッチする場合。

  - 反転時(negated 属性が true の場合)
    検証対象となる文字列が指定されたパターンと正規表現オプションによって表される正規表現にマッチしない場合。
    検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。

表 31 構成クラス

項番	クラス名	説明
1	RegexValidatorEx	正規表現による入力値検証を行う検証クラス
2	RegexValidatorExData	設定情報から RegexValidatorEx を生成、初期化するクラス

表 32 固有設定項目

項番	属性名	必須	解説	デフォルト値
1	pattern	○	正規表現のパターン	なし
2	options	-	正規表現オプション(RegexOptions 列挙体) を指定する。列挙体の値は以下の値の一つまたは複数の組み合わせとなる ・None ・IgnoreCase ・Multiline ・ExplicitCapture ・Compiled ・Singleline ・IgnorePatternWhiteSpace ・RighttoLeft 複数の値を指定する場合、コンマで区切って記述する	None
4	patternResourceName	-	正規表現パターンを読み込む際に利用するリソース名	なし
5	patternResourceTypeName	-	正規表現のパターンを読み込むリソースの型名	なし

表 33 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	正規表現のパターン
2	{4}	利用された正規表現オプション

以下に設定例を示す。

```
<validator pattern="^TEL [0-9][0-9][0-9][0-9]-?[0-9][0-9]-?[0-9][0-9][0-9][0-9]$"
  options="Compiled, IgnorePatternWhitespace"
  patternResourceName="" patternResourceType="" messageTemplate=""
  messageTemplateResourceName="" messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.RegexValidatorEx,
  TERASOLUNA.Fw.Common"
  name="Regex Tel Number Validator" />
```

リスト 10 設定例

- 必須入力チェック
  検証対象値が未入力でないかを検証する。フレームワークで提供する RequiredValidator を利用する。

  以下に、検証成功となるパターンを示す。

  - 通常時
    検証対象が null ではなく、検証対象値の文字列表現からホワイトスペース(半角空白、全角空白、改行、タブ文字)を取り除いた文字列の長さが 0 より大きい場合。

  - 反転時(negated 属性が true の場合)
    検証対象が null であるか、または検証対象値の文字列表現からホワイトスペース(半角空白、全角空白、改行、タブ文字)を取り除いた文字列の長さが 0 の場合。

  フレームワークで提供する入力値検証ルールは原則として null または空文字列に対して検証成功を返す。negated が true である場合も同様となる。null または空文字列を許容しない場合には、必須入力チェックを併用する。

表 34 構成クラス

項番	クラス名	説明
1	RequiredValidator	必須入力チェックを行う検証クラス
2	RequiredValidatorData	必須入力チェックを行う RequiredValidator を生成するクラス

必須入力チェックには、特に固有の設定項目はない。

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.RequiredValidator,
        TERASOLUNA.Fw.Common"
  name="Required Validator" />
```

リスト 11 設定例

- 数値文字列チェック
  検証対象の文字列表現が、半角数字（0～9)のみで構成されているか検証する。フレームワークで提供する NumericStringValidator を利用する。

  以下に、検証成功となるパターンを示す。

  - 通常時
    検証対象の文字列表現が半角数字(0～9)のみで構成されている場合。

  - 反転時(negated 属性が true の場合)
    検証対象の文字列表現に半角数字(0～9)以外の文字が含まれている場合。

    検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。

表 35 構成クラス

項番	クラス名	説明
1	NumericStringValidator	数値文字列チェックを行う検証クラス
2	NumericStringValidatorData	数値文字列チェックを行う NumericStringValidator を生成するクラス

数値文字列チェックには、特に固有の設定項目はない。

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.NumericStringValidator,
        TERASOLUNA.Fw.Common"
  name="Numeric String Validator" />
```

リスト 12 設定例

- 半角英数大文字列チェック

検証対象の文字列表現が、半角英数大文字のみで構成されているか検証する。フレームワークで提供する CapAlphaNumericStringValidator を利用する。半角英数大文字とは、半角数字(0～9)と半角大文字の英字(A～Z)を併せた文字集合である。

検証成功となるパターンを示す。

- 通常時
  検証対象の文字列表現が半角英数大文字のみで構成されている場合。
- 反転時(negated 属性が true の場合)
  検証対象の文字列表現に半角英数大文字以外の文字が含まれている場合。

検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。

表 36 構成クラス

項番	クラス名	説明
1	CapAlphaNumericStringValidator	半角英数大文字列チェックを行う検証クラス
2	CapAlphaNumericStringValidatorData	半角英数大文字列チェックを行う CapAlphaNumericStringValidator を生成するクラス

半角英数大文字列チェックには、特に固有の設定項目はない。

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.
        CapAlphaNumericStringValidator, TERASOLUNA.Fw.Common"
  name="Cap Alpha Numeric String Validator" />
```

リスト 13 設定例

- 半角英数文字列チェック

検証対象の文字列表現が、半角英数文字のみで構成されているか検証する。フレームワークで提供する AlphaNumericStringValidator を利用する。半角英数文字とは、半角数字(0～9)と半角の英字(a～z、A～Z)を併せた文字集合である。

以下に、検証成功となるパターンを示す。

- 通常時
  検証対象の文字列表現が半角英数文字のみで構成されている場合。

- 反転時(negated 属性が true の場合)
  検証対象の文字列表現に半角英数文字以外の文字が含まれている場合。

検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。

表 37 構成クラス

項番	クラス名	説明
1	AlphaNumericStringValidator	半角英数文字列チェックを行う検証クラス
2	AlphaNumericStringValidatorData	半角英数文字列チェックを行う AlphaNumericStringValidator を生成するクラス

半角英数文字列チェックには、特に固有の設定項目はない。

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.AlphaNumericStringValidator,
        TERASOLUNA.Fw.Common"
  name="Alpha Numeric String Validator" />
```

リスト 14 設定例

- 半角文字列チェック
  検証対象の文字列表現が、半角文字列のみで構成されているか検証する。フレームワ

ークで提供する HankakuStringValidator を利用する。半角文字とは、“＼¢ £ §¨ ¬°±´¶×÷”を除く unicode の 0 から 255 番目まで(拡張 ASCII コードの範囲)の文字に半角カナを加えた文字集合である。

以下に、検証成功となるパターンを示す。

- 通常時
  検証対象の文字列表現が半角文字のみで構成されている場合。

- 反転時(negated 属性が true の場合)
  検証対象の文字列表現に半角文字以外の文字が含まれている場合。

検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。

表 38 構成クラス

項番	クラス名	説明
1	HankakuStringValidator	半角文字列チェックを行う検証クラス
2	HankakuStringValidatorData	半角文字列チェックを行う HankakuStringValidator を生成するクラス

半角文字列チェックには、特に固有の設定項目はない。

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.HankakuStringValidator,
        TERASOLUNA.Fw.Common"
  name="Hankaku String Validator" />
```

リスト 15 設定例

,

- 半角カナ文字列チェック

  検証対象の文字列表現が、半角カナのみで構成されているか検証する。フレームワークで提供する HankakuKanaStringValidator を利用する。半角カナ文字とは、以下に示す文字集合である。"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ﾟｰ･､｡｢｣"

  以下に、検証成功となるパターンを示す。

  - 通常時
    検証対象の文字列表現が半角カナ文字のみで構成されている場合。
  - 反転時(negated 属性が true の場合)
    検証対象の文字列表現に半角カナ文字以外の文字が含まれている場合。

  検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。

表 39 構成クラス

項番	クラス名	説明
1	HankakuKanaStringValidator	半角カナ文字列チェックを行う検証クラス
2	HankakuKanaStringValidatorData	半角カナ文字列チェックを行う HankakuKanaStringValidator を生成するクラス

半角カナ文字列チェックには、特に固有の設定項目はない。

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.HankakuKanaStringValidator,
        TERASOLUNA.Fw.Common"
  name="HankakuKana String Validator" />
```

リスト 16 設定例

- 全角カナ文字列チェック

  検証対象の文字列表現が、全角カナ文字列のみで構成されているか検証する。フレームワークで提供する ZenkakuKanaStringValidator を利用する。全角カナ文字とは、以下に示す文字集合である。"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"

  以下に、検証成功となるパターンを示す。

  - 通常時
    検証対象の文字列表現が全角カナ文字のみで構成されている場合。
  - 反転時(negated 属性が true の場合)
    検証対象の文字列表現に全角カナ文字以外の文字が含まれている場合。

    検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。

表 40 構成クラス

項番	クラス名	説明
1	ZenkakuKanaStringValidator	全角カナ文字列チェックを行う Validator
2	ZenkakuKanaStringValidatorData	全角カナ文字列チェックを行う ZenkakuKanaStringValidator を生成するクラス

全角カナ文字列チェックには、特に固有の設定項目はない。

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.ZenkakuKanaStringValidator,
        TERASOLUNA.Fw.Common"
  name="Zenkaku Kana String Validator" />
```

リスト 17 設定例

- 全角文字列チェック

  検証対象の文字列表現が、全角文字のみで構成されているか検証する。フレームワークで提供する ZenkakuStringValidator を利用する。全角文字列とは、半角文字(半角文字列チェックの定義に準じる)ではない文字のみで構成された文字列のことである。

  以下に、検証成功となるパターンを示す。

  - 通常時
    検証対象の文字列表現が全角文字のみで構成されている場合。
  - 反転時(negated 属性が true の場合)
    検証対象の文字列表現に全角文字以外の文字が含まれている場合。

  検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。

表 41 構成クラス

項番	クラス名	説明
1	ZenkakuStringValidator	全角文字列チェックを行う検証クラス
2	ZenkakuStringValidatorData	全角文字列チェックを行う ZenkakuStringValidator を生成するクラス

全角文字列チェックには、特に固有の設定項目はない。

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.ZenkakuStringValidator,
        TERASOLUNA.Fw.Common"
  name="ZenkakuString String Validator" />
```

リスト 18 設定例

- URL 形式文字列チェック
  検証対象の文字列表現が、URL 形式の文字列であるかを検証する。フレームワークで提供する UrlValidator を利用する。URL の形式は、以下に示す。

  <プロトコル>://<ホスト名>:<ポート番号>/<パス文字列>

表 42 URL 形式文字列の構成要素

項番	名称	説明
1	プロトコル	http or https のみ対応。小文字と大文字は区別しない
2	ホスト名	“/“及び”:“以外の任意の文字列である
3	ポート番号	任意桁数の数値。省略可能であり、省略時は”:”を記述しないこと
4	パス	任意の文字列

  検証成功となるパターンを示す。

  - 通常時
    検証対象の文字列表現が URL 形式である場合。
  - 反転時(negated 属性が true の場合)
    検証対象の文字列表現が URL 形式ではない場合。

  検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。

表 43 構成クラス

項番	クラス名	説明
1	UrlValidator	URL 形式文字列チェックを行う検証クラス
2	UrlValidatorData	URL 形式文字列チェックを行う UrlValidator を生成するクラス

  URL 形式文字列チェックには、特に固有の設定項目はない。

  以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.UrlValidator,
        TERASOLUNA.Fw.Common"
  name="Url Format String Validator" />
```

リスト 19 設定例

- 日付形式チェック

  検証対象の文字列表現が、指定した形式の日付文字列であるかを検証する。フレームワークで提供する DateTimeFormatValidator を利用する。日付文字列の形式は、DateTimeFormatInfo で利用可能な形式指定文字列の組み合わせで指定する。たとえば、”yyyy/MM/dd”を形式として指定する場合、”2005/08/12”などの日付文字列には検証に成功するが、”2005-08-12”や”2005 年 08 月 12 日”などの文字列には検証に失敗する。

  検証成功となるパターンを示す。

  - 通常時
    検証対象の文字列表現が、指定した日付書式文字列に従って DateTime 型に変換可能である場合。
  - 反転時(negated 属性が true の場合)
    検証対象の文字列表現が、指定した日付書式文字列に従って DateTime 型に変換可能でない場合。

  検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。

表 44 構成クラス

項番	クラス名	説明
1	DateTimeFormatValidator	日付形式チェックを行う検証クラス
2	DateTimeFormatValidatorData	日付形式チェックを行う DateTimeFormatValidator を生成するクラス

表 45 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	dateTimeFormat	○	検証対象である日付形式文字列を指定する	なし

表 46 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	日付形式文字列。(dateTimeFormat)

以下に設定ファイルへの設定例を示す。

```
<validator negated="false" messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  dateTimeFormat="yyyy/MM/dd"
  type="TERASOLUNA.Fw.Common.Validation.Validators.DateTimeFormatValidator,
        TERASOLUNA.Fw"
  name="Date Time Format String Validator" />
```

リスト 20 設定

- int 型範囲チェック
  検証対象が int(System.Int32)型の数値または int 型に変換可能な文字列であり、指定した範囲に含まれているか検証する。フレームワークで提供する IntRangeValidator を利用する。

  検証成功となるパターンを示す。

  - 通常時
    検証対象が int 型の値または int 型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれる場合。
  - 反転時(negated 属性が true の場合)
    検証対象が int 型の値または int 型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれない場合。

  検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。

  文字列長チェックで示した範囲チェックについての注意点も参考のこと。

表 47 構成クラス

項番	クラス名	説明
1	IntRangeValidator	int 型範囲チェックを行う検証クラス
2	IntRangeValidatorData	int 型範囲チェックを行う IntRangeValidator を生成するクラス

表 48 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	lowerBound	-	検証対象である int 型の値の最小値を指定する	0
2	lowerBoundType	-	lowerBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Ignore
3	upperBound	-	検証対象である int 型の値の最大値を指定する	0
4	upperBoundType	-	upperBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Inclusive

表 49 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	最小値(lowerBound)
2	{4}	最小値の値の扱い(lowerBoundType)
3	{5}	最大値(upperBound)
4	{6}	最大値の値の扱い(upperBoundType)

なお、IntRangeValidator に限らず、Validation AB を利用する範囲チェックの仕組みでは、lowerBound(最小値)、lowerBoundType(指定した最小値の扱い)、upperBound(最大値)、upperBoundType(指定した最大値の扱い)の四つの属性を利用して範囲を指定する。共通して以下の制約があるので注意すること。

- lowerBound <= upperBound である必要がある
- lowerBoundType が Inclusive または Exclusive と指定されているとき、lowerBound は必ず指定する必要がある(lowerBound のデフォルト値がある場合を除く)
- upperBoundType が Inclusive または Exclusive と指定されているとき、upperBound は必ず指定する必要がある(upperBound のデフォルト値がある場合を除く)
- lowerBoundType と upperBoundType の両方が Ignore であってはならない
- lowerBound、upperBound に指定する文字列は検証しようとするデータの型に変換できなければならない

検証しようとするデータとは検証対象の値とは異なる。IntRangeValidator であれば、数値であるから、int 型であるが、後述する StringLengthRangeValidator では文字列を対象とするものの検証される値は文字列長であるから int 型である。
negated を true と指定した場合、有効となる範囲が反転する。このとき、Inclusive

/Exclusive の意味も逆転することに注意する。

以下に設定ファイルへの設定例を示す。

```
<validator lowerBound="10" lowerBoundType="Ignore" upperBound="20"
  upperBoundType="Exclusive" negated="false" messageTemplate=""
  messageTemplateResourceName="" messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.IntRangeValidator,
        TERASOLUNA.Fw.Common"
  name="Int Range Validator" />
```

リスト 21 設定例

- decimal 型範囲チェック

  検証対象が decimal(System.Decimal)型の数値または decimal 型に変換可能な文字列であり、指定した範囲に含まれているか検証する。フレームワークで提供する DecimalRangeValidator を利用する。

  検証成功となるパターンを示す。

  - 通常時
    検証対象が decimal 型の値または decimal 型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれる場合。
  - 反転時(negated 属性が true の場合)
    検証対象が decimal 型の値または decimal 型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれない場合。

  検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。文字列長チェックで示した範囲チェックについての注意点も参考のこと。

表 50 構成クラス

項番	クラス名	説明
1	DecimalRangeValidator	decimal 型範囲チェックを行う検証クラス
2	DecimalRangeValidatorData	decimal 型範囲チェックを行う DecimalRangeValidator を生成するクラス

表 51 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	lowerBound	-	検証対象である decimal 型の値の最小値を指定する	0
2	lowerBoundType	-	lowerBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Ignore
3	upperBound	-	検証対象である decimal 型の値の最大値を指定する	0
4	upperBoundType	-	upperBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Inclusive

表 52 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	最小値(lowerBound)
2	{4}	最小値の値の扱い(lowerBoundType)
3	{5}	最大値(upperBound)
4	{6}	最大値の値の扱い(upperBoundType)

以下に設定ファイルへの設定例を示す。

```
<validator lowerBound="0.1" lowerBoundType="Exclusive "
upperBound="100000000000000000000000000000"
  upperBoundType="Exclusive" negated="false" messageTemplate=""
  messageTemplateResourceName="" messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.DecimalRangeValidator,
        TERASOLUNA.Fw.Common"
  name="DecimalRange Validator" />
```

リスト 22 設定例

- float 型範囲チェック

  検証対象が float(System.Single)型の数値または float 型に変換可能な文字列であり、指定した範囲に含まれているか検証する。フレームワークで提供する FloatRangeValidator を利用する。

  検証成功となるパターンを示す。

  - 通常時

    検証対象が float 型の値または float 型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれる場合。
  - 反転時(negated 属性が true の場合)

    検証対象が float 型の値または float 型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれない場合。

  検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。文字列長チェックで示した範囲チェックについての注意点も参考のこと。

表 53 構成クラス

項番	クラス名	説明
1	FloatRangeValidator	float 型範囲チェックを行う検証クラス
2	FloatRangeValidatorData	float 型範囲チェックを行う FloatRangeValidator を生成するクラス

表 54 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	lowerBound	-	検証対象である float 型の値の最小値を指定する	0
2	lowerBoundType	-	lowerBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Ignore
3	upperBound	-	検証対象である float 型の値の最大値を指定する	0
4	upperBoundType	-	upperBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Inclusive

表 55 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	最小値(lowerBound)
2	{4}	最小値の値の扱い(lowerBoundType)
3	{5}	最大値(upperBound)
4	{6}	最大値の値の扱い(upperBoundType)

以下に設定ファイルへの設定例を示す。

```
<validator lowerBound="-0.1" lowerBoundType="Exclusive" upperBound="0.01"
  upperBoundType="Exclusive" negated="false" messageTemplate=""
  messageTemplateResourceName="" messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.FloatRangeValidator,
        TERASOLUNA.Fw.Common"
  name="Float Range Validator" />
```

リスト 23 設定例

- double 型範囲チェック
  検証対象が double(System.Double)型の数値または double 型に変換可能な文字列であり、指定した範囲に含まれているか検証する。フレームワークで提供する DoubleRangeValidator を利用する。

  検証成功となるパターンを示す。

  - 通常時
    検証対象が double 型の値または double 型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれる場合。
  - 反転時(negated 属性が true の場合)
    検証対象が double 型の値または double 型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれない場合。

  検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。文字列長チェックで示した範囲チェックについての注意点も参考のこと。

表 56 構成クラス

項番	クラス名	説明
1	DoubleRangeValidator	double 型範囲チェックを行う検証クラス
2	DoubleRangeValidatorData	double 型範囲チェックを行う DoubleRangeValidator を生成するクラス

表 57 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	lowerBound	-	検証対象である double 型の値の最小値を指定する	0
2	lowerBoundType	-	lowerBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Ignore
3	upperBound	-	検証対象である double 型の値の最大値を指定する	0
4	upperBoundType	-	upperBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Inclusive

表 58 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	最小値(lowerBound)
2	{4}	最小値の値の扱い(lowerBoundType)
3	{5}	最大値(upperBound)
4	{6}	最大値の値の扱い(upperBoundType)

以下に設定ファイルへの設定例を示す。

```
<validator lowerBound="10" lowerBoundType="Exclusive" upperBound="20"
  upperBoundType="Exclusive" negated="false" messageTemplate=""
  messageTemplateResourceName="" messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.DoubleRangeValidator,
        TERASOLUNA.Fw.Common"
  name="Double Range Validator" />
```

リスト 24 設定例

- 日付型範囲チェック

  検証対象が日付型(System.DateTime)型の数値または日付型に変換可能な文字列であり、指定した範囲に含まれているか検証する。フレームワークで提供するDateTimeRangeValidatorEx を利用する。Validation AB で提供されるDateTimeRangeValidator とは継承関係はなく、名前空間も異なるが、記述上の可読性を考慮してクラス名には"Ex"サフィックスを付与している。

  日付型範囲チェックでは、日付型に変換可能な文字列に対する範囲チェックも可能であるが、記述形式を指定することはできない。DateTimeConverter が対応した形式であれば検証対象となる。日付文字列の形式を指定するには、フレームワークで提供する日付文字列フォーマットチェック(DateTimeFormatValidator)と併用する。

  検証成功となるパターンを示す。

  - 通常時
    検証対象が日付型の値または日付型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれる場合。
  - 反転時(negated 属性が true の場合)
    検証対象が日付型の値または日付型の値に変換可能な文字列であり、かつ、最小値と最大値の間に含まれない場合。

  検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。文字列長チェックで示した範囲チェックについての注意点も参考のこと。

表 59 構成クラス

項番	クラス名	説明
1	DateTimeRangeValidator	日付型範囲チェックを行う検証クラス
2	DateTimeRangeValidatorData	日付型範囲チェックを行う DateTimeRangeValidator を生成するクラス

表 60 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	lowerBound	-	検証対象である DateTime 型の値の最小値を指定する	0 (1 年 1 月 1 日 00:00)
2	lowerBoundType	-	lowerBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Ignore
3	upperBound	-	検証対象である DateTime 型の値の最大値を指定する	0 (1 年 1 月 1 日 00:00)
4	upperBoundType	-	upperBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Inclusive

表 61 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	最小値(lowerBound)
2	{4}	最小値の値の扱い(lowerBoundType)
3	{5}	最大値(upperBound)
4	{6}	最大値の値の扱い(upperBoundType)

以下に設定ファイルへの設定例を示す。

```
<validator lowerBound="2007/04/01" lowerBoundType="Exclusive"
  upperBound="2008/04/01" upperBoundType="Exclusive"
  negated="false" messageTemplate=""
  messageTemplateResourceName="" messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.DateTimeRangeValidator,
        TERASOLUNA.Fw.Common"
  name="Float Range Validator" />
```

リスト 25 設定例

- byte 列長範囲チェック

  検証対象の文字列を指定したエンコード形式でバイト列にデコードした際、バイト列長が指定した範囲であるかどうかを検証する。フレームワークで提供する ByteRangeValidator を利用する。

  検証成功となるパターンを示す。

  - 通常時
    検証対象が文字列であり、かつ、指定したエンコーディングでデコードしたバイト列の長さが指定した最小値と最大値の間に含まれる場合。
  - 反転時(negated 属性が true の場合)
    検証対象が文字列であり、かつ、指定したエンコーディングでデコードしたバイト列の長さが指定した最小値と最大値の間に含まれない場合。

  検証対象が null である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。

表 62 構成クラス

項番	クラス名	説明
1	ByteRangeValidator	byte 列長範囲チェックを行う検証クラス
2	ByteRangeValidatorData	byte 列長範囲チェックを行う ByteRangeValidator を生成するクラス

表 63 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	maxByteLength	-	検証対象であるバイト長の最大値を int 型の値で指定する。範囲には指定した値自体が含まれる。	2,147,483,647
2	minByteLength	-	検証対象であるバイト長の最小値を int 型の値で指定する。範囲には指定した値自体が含まれる	0

3	encodingName	-	検証対象文字列をバイト列にデコードするために利用するエンコーディング名。コードページでの指定はできません。利用可能なエンコーディング名はSystem.Text.Encoding クラスでサポートされているエンコーディングを参照のこと。	utf-8

表 64 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	バイト列長の最大値。(maxByteLength)
2	{4}	バイト列長の最小値。(minByteLength)
3	{5}	エンコーディング名

以下に設定ファイルへの設定例を示す。

```
<validator minByteLength="5" maxByteLength="10"  encodingName="iso-2022-jp"
  negated="false" messageTemplate=""
  messageTemplateResourceName="" messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.ByteRangeValidator,
        TERASOLUNA.Fw.Common"
  name="Byte Range Validator" />
```

リスト 26 設定例

- 文字列長チェック

  検証対象の文字列表現の文字列長が指定した範囲であるかどうかを検証する。フレームワークで提供する StringLengthRangeValidator を利用する。

  最小値を Ignore(無視)指定することで最大文字数制限として、最大値を Ignore 指定することで最小文字数制限として機能する。また、最大値と最小値として同じ値を指定し、いずれも Inclusive(値を含む)として指定すれば、固定文字数チェックとして機能する。

  検証成功となるパターンを示す。

  - 通常時
    検証対象の文字列表現の文字列長が最小値と最大値の間に含まれる場合。
  - 反転時(negated 属性が true の場合)
    検証対象の文字列表現の文字列長が最小値と最大値の間に含まれない場合。

  検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。

表 65 構成クラス

項番	クラス名	説明
1	StringLengthRangeValidator	文字列の長さが指定した範囲内であるかどうかを検証する検証クラス
2	StringLengthRanheValidatorData	設定情報から StringLengthRangeValidator を生成、初期化するクラス

表 66 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	lowerBound	-	検証対象である文字列長範囲の最小値を指定する	0
2	lowerBoundType	-	lowerBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体。以下の値のいずれかとなる。 ・Ignore (値を無視する) ・Inclusive (値を含む) ・Exclusive (値を含まない)	Ignore
3	upperBound	-	検証対象である文字列長範囲の最大値を指定する	0

4	upperBoundType	-	upperBound で指定した最小値の値の扱いを指定する RangeBoundaryType 列挙体	Inclusive

表 67 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	最小値(lowerBound)
2	{4}	最小値の値の扱い(lowerBoundType)
3	{5}	最大値(upperBound)
4	{6}	最大値の値の扱い(upperBoundType)

以下に設定ファイルへの設定例を示す。

```
<validator lowerBound="10" lowerBoundType="Inclusive" upperBound="20"
  upperBoundType="Inclusive" negated="false"
  messageTemplate="値は10文字以上、20文字以下で指定してください。"
  messageTemplateResourceName="" messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.StringLengthRangeValidator,
  TERASOLUNA.Fw.Common"
  name="String Length Range Validator" />
```

リスト 27 設定例

- 必須文字列チェック(禁止文字列チェック)
  検証対象文字列が、指定した文字を含んでいるかどうかの検証を行う。フレームワークで提供する ContainsCharactersValidatorEx を利用する。

  negated 属性を省略または false と指定した場合、characterSet 属性で指定した文字が含まれていない場合に検証エラーとなる。negated 属性を true と指定することで、指定した文字が含まれる場合に検証エラーとする禁止文字列チェックとして利用することができる。

  検証成功となるパターンを示す。

  - 通常時
    検証対象が文字列であり、かつ、containsCharacter で「Any」が指定されている場合には、characterSet で指定された文字のうちいずれかを含んでいれば検証成功となる。containsCharacter で「All」が指定されている場合には、characterSet で指定された文字を全て含んでいれば検証成功となる。
  - 反転時(negated 属性が true の場合)
    検証対象が文字列であり、かつ、containsCharacter で「Any」が指定されている場合には、characterSet で指定された文字が一つも含まれていなければ検証成功となる。containsCharacter で「All」が指定されている場合には、characterSet で指定された文字を全て含んでいる場合以外に検証成功となる。

  検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。

表 68 構成クラス

項番	クラス名	説明
1	ContainsCharactersValidatorEx	指定した文字が含まれているかどうかを検証する検証クラス
2	ContainsCharactersValidatorExData	設定情報から ContainsCharactersValidator を生成、初期化するクラス

表 69 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	characterSet	○	検証対象の文字列中で存在をチェックする文字。(複数指定する場合、コンマなどで区切らず、"AB."と続けて記述する。)	なし
2	containsCharacter	○	characterSet で指定した文字の存在をチェックする方式を指定する ContainsCharacters 列挙体。 以下の値のいずれかとなる。 ・Any (いずれか一つを含む) ・All (全ての文字を含む)	Any

表 70 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	チェック対象の文字。(characterSet)
2	{4}	文字のチェック方式。(containsCharacter)

以下に設定ファイルへの設定例を示す。

```
<validator characterSet="qwerty" containsCharacter="All" negated="true"
  messageTemplate="" messageTemplateResourceName=""
  messageTemplateResourceType="" tag=""
  type="TERASOLUNA.Fw.Common.Validation.Validators.ContainsCharactersValidatorEx,
  TERASOLUNA.Fw.Common"
  name="Contains Characters Ex Validator" />
```

リスト 28 設定例

- 数値チェック

  検証対象の文字列表現が、整数部と小数部がそれぞれ指定した桁数である数値の形式であるかを検証する。フレームワークで提供する NumberValidator を利用する。桁数のチェックには、指定した値と等しいか調べる一致チェックと、指定した以内の範囲に含まれるか調べる範囲チェックを利用することができる。

  検証成功となるパターンを示す。

  - 通常時
    検証対象の文字列表現が整数または小数の数値形式であり、桁数がそれぞれ指定した値・範囲に含まれる場合。
  - 反転時(negated 属性が true の場合)
    検証対象の文字列表現が整数または小数の数値形式でない場合、または、桁数がそれぞれ指定した値・範囲に含まれない場合。

  検証対象が null または空文字列である場合、negated の値に関わらず、検証処理は実行されない。検証結果は必ず成功となる。null または空文字列を許容しない場合には、フレームワークで提供する必須入力チェック(RequiredValidator)と併用する。

表 71 構成クラス

項番	クラス名	説明
1	NumberValidator	数値チェックを行う検証クラス
2	NumberValidator	数値チェックを行う NumberValidator を生成するクラス

表 72 固有設定項目

項番	属性名	必須	説明	デフォルト値
1	integerLength	○	数値文字列の整数部の桁数を int 型の自然数で指定する。最小値は 1	なし
2	scale	-	数値文字列の小数部の桁数を int 型の値で指定する。最小値は 0 であり、0 を指定した際には整数チェックとなる	0

3	isAccordedInteger	-	整数部の桁数一致チェックを行うかどうかを指定する。True であれば、integerLength で指定した桁数と一致しているかどうかを検証する。false であれば、指定した桁数以下であるか検証する	false
4	isAccordedScale	-	isAccordedInteger と同様に小数部の桁数一致チェックを行うかどうかを指定する	false

表 73 メッセージテンプレート文字列

項番	プレースホルダ	説明
1	{3}	指定した整数部の桁数。(integerLength)
2	{4}	指定した小数部の桁数(scale)

以下に設定ファイルへの設定例を示す。

```
<validator integerLength="5" scale="4"
 isAccordedInteger="false" isAccordedScale="true"
 negated="false" messageTemplate=""
 messageTemplateResourceName="" messageTemplateResourceType="" tag=""
 type="TERASOLUNA.Fw.Common.Validation.Validators.NumberValidator,
       TERASOLUNA.Fw.Common"
 name="Number Validator" />
```

リスト 29 設定例

■ 内部構成

◆ 構成クラス

表 74 構成クラス一覧

項番	クラス名	説明
1	AlphaNumericStringValidator	半角英数文字列チェックを行う検証クラス
2	AlphaNumericStringValidatorData	AlphaNumericStringValidator の生成に利用するクラス
3	ByteRangeValidator	エンコーディングを指定して文字列のバイト長範囲チェックを行う検証クラス
4	ByteRangeValidatorData	ByteRangeValidator の生成に利用するクラス
5	CapAlphaNumericStringValidator	半角英数大文字列チェックを行う検証クラス
6	CapAlphaNumericStringValidatorData	CapAlphaNumericStringValidator の生成に利用するクラス
7	CaseCheckUtil	文字種・パターンチェックを行う Validator で利用するユーティリティクラス
8	ContainsCharactersValidatorEx	指定した文字列を含んでいるかどうかを検証する
9	ContainsCharactersValidatorExData	ContainsCharactersValidatorEx の生成に利用するクラス
10	DateTimeFormatValidator	日付形式文字列チェックを行う検証クラス
11	DateTimeFormatValidatorData	DateTimeFormatValidator の生成に利用するクラス
12	DateTimeRangeValidatorEx	日付の範囲チェックを行う検証クラス
13	DateTimeRangeValidatorExData	DateTimeRangeValidatorEx の生成に利用するクラス
14	DecimalRangeValidator	Decimal 値の範囲チェックを行う検証クラス
15	DecimalRangeValidatorData	DecimalRangeValidator の生成に利用するクラス
16	DoubleRangeValidator	Double 値の範囲チェックを行う検証クラス
17	DoubleRangeValidatorData	DoubleRangeValidator の生成に利用するクラス
18	FloatRangeValidator	Float 値の範囲チェックを行う検証クラス

19	FloatRangeValidatorData	FloatRangeValidator の生成に利用するクラス
20	HankakuKanaStringValidator	半角カナ文字列チェックを行う検証クラス
21	HankakuKanaStringValidatorData	HankakuKanaStringValidator の生成に利用するクラス
22	HankakuStringValidator	半角文字列チェックを行う検証クラス
23	HankakuStringValidatorData	HankakuStringValidator の生成に利用するクラス
24	IntRangeValidator	Int 値の範囲チェックを行う検証クラス
25	IntRangeValidatorData	IntRangeValidator の生成に利用するクラス
26	NumberRangeValidator	数値型の範囲チェックを行う Validator の基底クラス
27	NumberValidator	数値の桁数チェックを行う検証クラス
28	NumberValidatorData	NumberValidator の生成に利用するクラス
29	NumericStringValidator	数値文字列チェックを行う検証クラス
30	NumericStringValidatorData	NumericStringValidator の生成に利用するクラス
31	RequiredValidator	必須チェックを行う検証クラス
32	RequiredValidatorData	RequiredValidator の生成に利用するクラス
33	RegexValidatorEx	正規表現チェックを行う検証クラス
34	RegexValidatorExDaya	RegexValidatorEx の生成に利用するクラス
35	StringLengthRangeValidator	文字列長チェックを行う検証クラス
36	StringLengthRangeValidatorData	StringLengthRangeValidator の生成に利用するクラス
37	TypeRangeValidator	型指定した範囲チェックを行う Validator の基底クラス
38	UrlValidator	URL 形式文字列チェックを行う検証クラス
39	UrlValidatorData	UrlValidator の生成に利用するクラス
40	VabValidator	Validation Application Block の仕組みを利用して入力値検証を行う IValidator 実装クラス。
41	ZenkakuKanaStringValidator	全角カナ文字列チェックを行う検証クラス

42	ZenkakuKanaStringValidatorData	ZenkakuKanaStringValidator の生成に利用するクラス
43	ZenkakuStringValidator	全角文字列チェックを行う検証クラス
44	ZenkakuStringValidatorData	ZenkakuStringValidator の生成に利用するクラス

■ 拡張ポイント

◆ 検証ルールを拡張する方法

一つの検証ルールは役割の異なる二種類のクラスによって実現される。実際に入力値検証を実施する Validator 検証クラスと、設定情報の解析と Validator インスタンスの初期化を行う ValidatorData クラスである。下表にそれぞれの違いを示す。

表 75 入力値検証クラスの構成

項番	名称	説明
1	ValidatorData クラス	Microsoft.Practices.EnterpriseLibrary.Validation.Configuration.ValidatorData を継承した、Validator を生成・初期化するためのクラス。設定情報から Validator を生成するための DoCreateValidator メソッドを実装する。
2	Validator クラス	Microsoft.Practices.EnterpriseLibrary.Validation.Validator を継承した入力値検証の実装クラス 検証を行う DoValidate メソッドを実装する。 ConfigurationElementType 属性をクラスに付与し、利用する ValidatorData クラスの型を指定する。

Validation AB は、設定情報から Validator の型を取得するが、その型のインスタンスを直接生成しない。利用される Validator が生成される過程を以下に示す。

(1) 設定情報から、利用する Validator の型(TypeA)を取得する。
(2) 取得した型 TypeA の ConfigurationElementType 属性より、Validator の生成・初期化を行う ValidatorData クラスの型(TypeB)を取得し、型 TypeB のインスタンスを作成する。
(3) 型 TypeB の CreateValidator メソッドを用いて、実際に利用する Validator のインスタンスを取得する。

◆ 実装例

数字文字列のみを許可するシンプルな Validator の実装例を示す。Validation AB の提供する正規表現チェックを拡張して実装し、対応する ValidatorData クラスを作成する。

```
namespace SampleValidator
{
    [ConfigurationElementType(typeof(NumberStringValidatorData))]
    public class NumberStringValidator : RegexValidator
    {
        /// <summary>
        /// 正規表現を利用するValidatorのコンストラクタでパターンを固定
        /// </summary>
        public NumberStringValidator(string messageTemplate, bool negated)
            : base(@"[0-9]*", messageTemplate, negated)
        {
        }
    }
}
```

リスト 30　Validator クラス

```
namespace SampleValidatorData
{
    public class NumberStringValidatorData : ValueValidatorData
    {
        public NumberStringValidatorData()
        { }
        protected override Validator DoCreateValidator(Type targetType)
        {
            // NumberStringValidatorを生成
            return new NumberStringValidator(MessageTemplate, Negated);
        }
    }
}
```

リスト 31　ValidatorData クラス

```
<validator negated="false"
   messageTemplate="{2}には数字のみを入力してください。{0}は数字以外を含んでいるか、
文字列ではありません。"
   messageTemplateResourceName="" messageTemplateResourceType=""
   tag="数値項目"
   type="SampleValidatorData.NumberStringValidator, SampleValidatorData"
   name="number validator test" />
```

リスト 32 設定ファイル

例では固有の設定項目を持たないため、SampleValidatorData は単に SampleValidator のインスタンスを生成する処理のみを行う。生成される SampleValidator は数字文字列を表す固定の正規表現を用いて入力値検証を行う、RegexValidator のサブクラスである。RegexValidator はコンストラクタの第一引数で正規表現パターン文字列を受け取るため、SampleValidator では、親クラスのコンストラクタに固定の正規表現パターンを渡すことで機能を実現している。

固有の設定項目を利用して Validator を初期化するには、validator 要素に属性として指定された設定情報を取得し、Validator に設定する。ValidatorData クラスのインデクサに属性名を指定することで、validator 要素に設定された属性の値を取得することができる。

入力値検証のロジックを独自に定義するには、Validator を実装する際、DoValidate メソッドをオーバーライドする。その際、ValueValidator を親クラスに持つ場合には、negated が true と指定されている場合の検証条件の反転を考慮すること。

◆ ファクトリクラスの変更

Validator のインスタンスを生成するために利用するファクトリクラスは、必要に応じて変更することができる。特に指定がないとき、ファクトリクラスは ValidatorFactory が用いられる。

ファクトリクラスを変更する場合は、構成ファイル[4]の appSettings 要素に、”ValidatorFactoryTypeName”をキーとして、利用するファクトリクラスのアセンブリ修飾名を記述する。なお、本機能説明書においては標準のファクトリクラスである ValidatorFactory を用いた場合の例のみを示す。

以下に構成ファイルの例を示す。

[4] 構成ファイル：App.config や WebConfig のことを示す

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="ValidatorFactoryTypeName"
         value="MyNamespace.MyValidatorFactory, MyNamespace"/>
  </appSettings>
</configuration>
```

リスト 33 ファクトリ型指定の例(構成ファイル)

◆ 入力値検証実行クラスの変更

標準のファクトリを利用する場合、特に指定がないとき入力値検証実行クラス(IValidator 実装クラス)は VabValidator が用いられる。

利用する IValidator 実装クラスを変更する場合は、構成ファイルの appSettings 要素に、"ValidatorTypeName"をキーとしてアセンブリ修飾名を記述する。この設定をもとに、入力値検証生成機能(ValidatorFactory クラス)によって入力値検証実行クラスのインスタンスが生成される。

以下に構成ファイルの例を示す。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="ValidatorTypeName"
         value="TERASOLUNA.Fw.Common.Validation.VabValidator,
                TERASOLUNA.Fw.Common" />
  </appSettings>
</configuration>
```

リスト 34 入力値検証実行クラス指定の例(構成ファイル)

■ 関連機能

- FB-01 イベント処理機能
- WB-01 リクエストコントローラ機能

CM-03 ログ出力機能

■ 概要

本機能では、アプリケーションで統一的にログを出力する仕組みを提供する。個別のロギングパッケージ[1]をラップするため、異なるロギングパッケージへ移行した場合も、コードの修正を最小限に抑えることができる。また、Jakarta Commons Logging[2]と同等のログレベルを備えているため、サーバ側が Java、クライアント側が.NET という組み合わせの開発においても、アプリケーション全体で統一的なログポリシーを定めることができる。

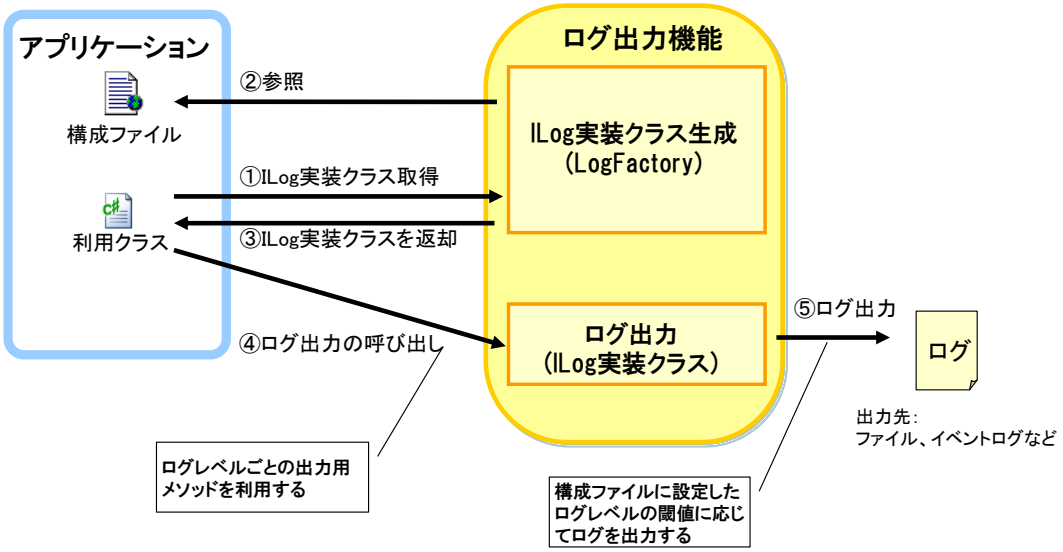


図 1 ログ出力機能

ログ出力機能を利用するクラスは、はじめにロガー(ILog 実装クラスのインスタンス)を取得する。ログ出力機能の内部では、構成ファイル[3]を参照して生成するロガーの種類や設定を決定する。ログを出力する際は、ロガーに用意されているログレベルごとのログ出力用メソッドを呼び出す。

本機能では、ロガーの標準実装として TraceSourceLogger を提供している。

---

[1] Logging AB :マイクロソフトが推進するオープンソース・ライブラリ「Enterprise Library」に含まれる、ログ出力用の Application Block [http://www.codeplex.com/entlib]
log4net :Apache プロジェクトでオープンソースとして開発されているロギングライブラリ。「log4j」を.NET に移植したもの [http://logging.apache.org/log4net/index.html]

[2] Jakarta Commons Logging :Apache プロジェクトでオープンソースとして開発されているロギングパッケージ間の互換性問題解決するためのコンポーネント[http://commons.apache.org/logging/]

[3] 構成ファイル:web.config や App.config のことを示す。

■ 使用方法

◆ 構成ファイル

TERASOLUNA が標準で提供する TraceSourceLogger を利用する場合、構成ファイルに TraceSource の設定をする。

表 1 構成ファイル

ノード	属性	必須	値
/configuration/system.diagnostics/			複数可
		○	詳細な設定方法は MSDN を参照

- TraceSource の設定

  TraceSourceLogger を使用する場合、TraceSource の設定を構成ファイルに記述する。TraceSourceLogger は、構成ファイルで定義した「カテゴリ」、カテゴリごとの「ログレベル」「出力先」に基づいて、ログ出力する機能を提供する。

  構成ファイルに設定する TraceSource のログレベルに関して、本機能で定義するログレベルと TraceSource のログレベルに違いがあるため、注意が必要である。構成ファイルに設定するログレベルの閾値と、有効なログレベルの対応を表 2 に示す。

表 2 ログレベルの閾値と有効なログレベルの対応表

		構成ファイルに設定するログレベルの閾値						
項番	ログレベル	Off	Critical	Error	Warning	Information	Verbose	All
1	FATAL		○	○	○	○	○	○
2	ERROR			○	○	○	○	○
3	WARN				○	○	○	○
4	INFO					○	○	○
5	DEBUG						○	○
6	TRACE							○

表 3 に示すような関心の異なるログを、別々のファイルに出力する場合の構成ファイルの設定例をリスト 1 に示す。

表 3 出力するログの種類

項番	ログの種類	カテゴリ名	ログレベルの閾値	出力先のファイル名
1	統計用のユーザによるボタン押下を単位としたログ	SubmitLog	TRACE	SubmitLog.txt
2	運用管理ソフトウェアの監視対象として利用するログ	FatalLog	FATAL	FatalLog.txt

3	その他のログ	Default	ERROR	DefaultLog.txt

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <!--使用するLogFactoryを指定する-->
    <add key="LogFactoryTypeName"
value="TERASOLUNA.Fw.Common.Logging.TraceSourceLog.TraceSourceLogFactory,TERA
SOLUNA.Fw.Common" />
  </appSettings>
  <system.diagnostics>
    <sources>
      <source name="SubmitLog" switchName="SubmitLogSwitch"
        switchType="System.Diagnostics.SourceSwitch">
        <listeners>
          <add name="SubmitLogFile"/>
          <remove name="Default"/>
        </listeners>
      </source>
      <source name="FatalLog" switchName="FatalLogSwitch"
        switchType="System.Diagnostics.SourceSwitch">
        <listeners>
          <add name="FatalLogFile"/>
          <remove name="Default"/>
        </listeners>
      </source>
      <source name="Default" switchName="DefaultLogSwitch"
        switchType="System.Diagnostics.SourceSwitch">
        <listeners>
          <add name="DefaultLogFile"/>
          <remove name="Default"/>
        </listeners>
      </source>
    </sources>
    <switches>
      <add name="SubmitLogSwitch" value="All"/>
      <add name="FatalLogSwitch" value="Critical"/>
      <add name="DefaultLogSwitch" value="Error"/>
    </switches>
    <sharedListeners>
      <add name="SubmitLogFile"
        type="System.Diagnostics.TextWriterTraceListener"
        initializeData="SubmitLog.txt">
      </add>
      <add name="FatalLogFile"
       type="System.Diagnostics.TextWriterTraceListener"
       initializeData="FatalLog.txt">
      </add>
      <add name="DefaultLogFile"
       type="System.Diagnostics.TextWriterTraceListener"
       initializeData="DefaultLog.txt">
      </add>
    </sharedListeners>
  </system.diagnostics>
</configuration>
```

カテゴリ名

ログの種類を定義

ログレベルの閾値

ログの種類ごとの
ログレベルを定義

出力先の種類

ログの種類ごとの
出力先を定義

出力先のファイル名

リスト 1 構成ファイル記述例

TraceSource の詳細な設定方法は MSDN を参照のこと。

◆ 実装方法

- ロガーの取得方法
  ロガーを取得するには LogFacotry クラスの GetLogger メソッドを呼び出す。GetLogger メソッドの一覧を表 4 に示す。

表 4　LogFactory クラスの GetLogger メソッド一覧

項番	GetLogger メソッドの種類	第 1 引数	第 2 引数	戻り値の型
1	public static ILog GetLogger(string className, params object[] args)	クラス名を表す文字列	object 型の可変長引数	ILog
2	public static ILog GetLogger(Type classType, params object[] args)	クラスの Type	object 型の可変長引数	ILog
3	public static T GetLogger<T>(string className, params object[] args) where T : class, ILog	クラス名を表す文字列	object 型の可変長引数	T（ジェネリック型）
4	public static T GetLogger<T>(Type classType, params object[] args) where T : class, ILog	クラスの Type	object 型の可変長引数	T（ジェネリック型）

TraceSourceLogger を使用する場合、GetLogger メソッドの第 2 引数には構成ファイルに定義したカテゴリ名を指定する。省略した場合は ”Default” になる。指定したカテゴリ名が構成ファイルに定義されていない場合、エラーにはならないが、ログレベルの閾値が Off となるためログは出力されない。また、第 3 引数以降の入力は無視される。

カテゴリ名を指定してロガーを取得する場合と、指定せずに取得する場合の実装例をリスト 2 に示す。

```
class Program
{
    // カテゴリ名（“CategoryName”）を指定してロガーを取得する
    private static ILog _loggerA = LogFactory.GetLogger(typeof(Program), “CategoryName”);
    // カテゴリ名を指定せずにロガーを取得する
    private static ILog _loggerB = LogFactory.GetLogger(typeof(Program));
    static void Main(string[] args)
    {
      // 処理
    }
}
```

第 2 引数にカテゴリ名を指定する。
省略した場合は “Default” になる。

リスト 2　ロガーを取得する実装例

- ログの出力方法

  ログを出力するには、ロガーのログ出力用メソッドを呼び出す。メソッドはログレベルごとに用意されており、引数にはメッセージや例外をとる。ログレベルと対応するメソッドの一覧を表 5 に示す。

表 5 ログレベルと対応するメソッド

項番	種類	ログレベル	ログ出力用メソッド	出力ログレベル確認用プロパティ
1	致命的なエラー	FATAL	Fatal(object message) Fatal(object message, Exception ex)	IsFatalEnabled
2	エラー	ERROR	Error(object message) Error(object message, Exception ex)	IsErrorEnabled
3	警告	WARN	Warn(object message) Warn(object message, Exception ex)	IsWarnEnabled
4	情報	INFO	Info(object message) Info(object message, Exception ex)	IsInfoEnabled
5	デバッグ	DEBUG	Debug(object message) Debug(object message, Exception ex)	IsDebugEnabled
6	トレース	TRACE	Trace(object message) Trace(object message, Exception ex)	IsTraceEnabled

引数が 1 つのメソッドでは、”message”に指定された引数をログメッセージとして出力する。引数が 2 つのメソッドでは、”message”に加えて”ex”に指定された例外を共に出力する。

出力ログレベル確認用のプロパティは、そのログレベルが現在有効かどうかをチェックし、有効なら true、無効なら false を返す。ログ出力用メソッドの引数に与えるメッセージを作成するために、文字列の連結のような重い処理を行う場合、このプロパティをチェックすることで、ログ出力のオーバーヘッドを減らすことができる。

※イベント ID を指定してログを出力する方法

イベント ID はデフォルトで 0 に設定されているが、TraceSourceLogger に実装されているログ出力用メソッドを利用することで、イベント ID を指定してログを出力することができる。このメソッドを呼び出す場合、ロガーを取得する GetLogger メソッドにジェネリックの型（ITraceSourceLog）を指定し、取得したロガーを ITraceSourceLog 型で受け取る。実装例をリスト 3 に示す。

```
class Program
{
    // ジェネリックの型を指定してロガーを取得する
    private static ITraceSourceLog logger =
      LogFactory.GetLogger<ITraceSourceLog>(typeof(Program));

    static void Main(string[] args)
    {
        if (logger.IsDebugEnabled)
        {
            // イベントID
            int eventId = 1;

            logger.Debug("出力するログメッセージ", eventId);
        }
    }
}
```

ジェネリックの型に ITraceSourceLog を指定

第 2 引数にイベント ID を指定するメソッドを呼び出す

リスト 3 ジェネリックの型を指定してロガーを取得する実装例

- 実装例

  表 3 に示した種類のログを、カテゴリごとに出力する場合の実装例をリスト 4 に示す。なお、カテゴリの定義、カテゴリごとのログレベルや出力先の定義はリスト 1 のように構成ファイルに設定されているものとする。

```
class Program
{
    // 統計用のユーザによるボタン押下を単位としたログ
    private static ILog _submitLog = LogFactory.GetLogger(typeof(Program), "SubmitLog");

    // 運用管理ソフトウェアの監視対象として利用するログ
    private static ILog _fatalLog = LogFactory.GetLogger(typeof(Program), "FatalLog");

    // その他のログ(カテゴリはDefault)
    private static ILog _defaultLog = LogFactory.GetLogger(typeof(Program));

    static void Main(string[] args)
    {
        // ログの出力
        if (_submitLog.IsTraceEnabled)
        {
            _submitLog.Trace("統計用のログメッセージ");
        }

        if (_fatalLog.IsFatalEnabled)
        {
            _fatalLog.Fatal("運用管理ソフトウェアの監視対象用のログメッセージ");
        }

        if (_defaultLog.IsFatalEnabled)
        {
            _defaultLog.Fatal("致命的エラー");
        }

        try
        {
            throw new Exception();
        }
        catch (Exception ex)
        {
            if (_defaultLog.IsErrorEnabled)
            {
                _defaultLog.Error("エラー", ex);
            }
        }

        if (_defaultLog.IsWarnEnabled)
        {
            _defaultLog.Warn("警告");
        }
    }
}
```

第 2 引数にカテゴリ名を指定する。
省略した場合は“Default" になる。

Defaultカテゴリはログレベルの
閾値が”ERROR”に設定されている
ため、FATALとERRORが出力される

WARNは出力されない

リスト 4 TraceSourceLog を使用する場合の実装例

- 出力結果例

  リスト 4 の実装例を実行した場合、ログの種類ごとにテキストファイルが作成される。それぞれの出力結果を以下に示す。

SubmitLog.txt

```
SubmitLog Verbose: 0 : TRACE 2008-06-27 20:14:58,058 [1] (ConsoleApplication1.Program) - 統計用のログメッセージ
```

FatalLog.txt

```
FatalLog Critical: 0 : FATAL 2008-06-27 20:14:58,058 [1] (ConsoleApplication1.Program) - 運用管理ソフトウェアの監視対象用のログメッセージ
```

DefaultLog.txt

```
Default Critical: 0 : FATAL 2008-06-27 20:14:58,073 [1] (ConsoleApplication1.Program) - 致命的エラー
Default Error: 0 : ERROR 2008-06-27 20:14:58,073 [1] (ConsoleApplication1.Program) - エラー
<System.Exception - 種類 'System.Exception' の例外がスローされました。>
System.Exception : 種類 'System.Exception' の例外がスローされました。
    場所 ConsoleApplication1.Program.Main(String[] args) 場所 C:¥Documents and Settings¥ユーザ名¥My Documents¥Visual Studio 2005¥Projects¥ConsoleApplication1¥ConsoleApplication1¥Program.cs:行 39
```

出力結果のフォーマット

```
{Category} {TraceEventType}:{EventID}:{LogLevel} {Date} [[ThreadID]] (ClassName) - {Message} <{Exception}>
Exception Detail
```

表 6 出力結果のフォーマットの要素

項番	要素名	説明
1	Category	カテゴリ名。
2	TraceEventType	TraceSourceLogger が内部で TraceSource を利用してログ出力する際に指定したイベントの種類。
3	EventID	イベント ID。デフォルトは 0。
4	LogLevel	ログレベル。
5	Date	現在の日付(yyyy-MM-dd HH:mm:ss,fff)
6	ThreadID	現在実行中のスレッドの ID。
7	ClassName	ログを出力したクラス名。ロガーを取得する GetLogger メソッドの第 1 引数に与えたクラス情報(クラスの Type もしくはクラス名を表す文字列)が使用される。
8	Message	ログ出力用メソッドの引数に与えられたメッセージ。
9	Exception	ログ出力用メソッドの引数に与えられた例外の型とメッセージ。
10	Exception Detail	例外の型とメッセージ。InnerException を含むスタックトレース。

■ 内部構成

本機能の内部構成について説明する。

● ログレベル

ログを出力する時、その優先度や重要度を考慮して「ログレベル」を設定する。本機能では、表 7 ログレベルの種類に示す 6 つのログレベルを用意している。

表 7 ログレベルの種類

項番	種類	ログレベル	説明
1	致命的なエラー	FATAL	システムに対して致命的な障害が発生した場合に利用する。
2	エラー	ERROR	予期せぬ動作などにより、正しく処理できない場合に利用する。通常は例外発生時に出力する。
3	警告	WARN	エラーの原因やバグの原因となりそうなときに利用する。例外をやむを得ず捕捉して、処理を業務フローに戻す場合などに出力する。
4	情報	INFO	システムの動作を通知する際に利用する。起動時や処理の開始など、重要な動作(外部仕様にかかわるもの)をトレースしたい場合に出力する。
5	デバッグ	DEBUG	起動時・メソッド呼び出し時・インスタンス生成時など、内部の動作をトレースしたい場合に利用する。
6	トレース	TRACE	詳細なデバッグ情報を出力する際に利用する。モジュール内部の情報、ループの繰り返しで大量に出力される情報など。

表 7 のログレベルは、Jakarta Commons Logging に合わせて定義したものである。TraceSource や Logging AB、log4net のログレベルとは表 8 のように対応している。コーディング時に特に意識する必要はないが、個別のロギングパッケージの設定をする場合に注意が必要である。

表 8 TERASOLUNA Logging とロギングパッケージのログレベルの対応

項番	TERASOLUNA Logging	ロギングパッケージ		
		TraceSource	Logging AB	Log4net
1	FATAL	Critical	Critical	FATAL
2	ERROR	Error	Error	ERROR
3	WARN	Warning	Warning	WARN
4	INFO	Information	Information	INFO
5	DEBUG	Verbose	Verbose	DEBUG
6	TRACE	※	※	※

※ロギングパッケージには TRACE にあたるログレベルが定義されていないため、個別の

Logger で定義する必要がある。標準実装の TraceSourceLogger における Trace レベルの定義は、「TraceSource の設定」を参照のこと。

◆ 構成クラス

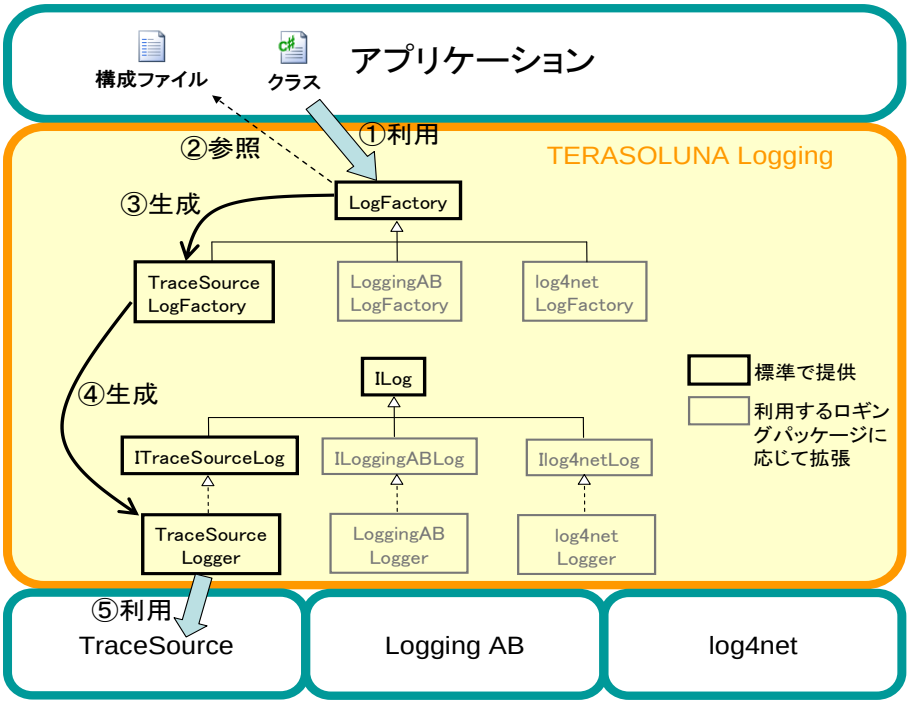


図 2 構成クラス図

表 9 構成クラス一覧

項番	クラス名	説明
1	ILog	ログ出力クラスに関わらず、統一的な方法でログを出力するためのインターフェイス。
2	LogFactory	ILog 実装クラスを生成するための抽象メソッドを持つ抽象クラス。構成ファイルに基づいて、ファクトリクラスを生成する。
3	ITraceSourceLog	ILog を継承し、TraceSource 用の抽象メソッドを追加したインターフェイス。
4	TraceSourceLogger	ITraceSourceLog を実装したログ出力クラス。TraceSource を使ってログを出力する。
5	TraceSourceLogFactory	TraceSourceLogger を生成するファクトリクラス。LogFactory を継承しており、TraceSourceLogger を生成するためのメソッドを実装している。
6	LogLevel	ログレベル列挙体を定義するクラス。

■ 拡張ポイント

Logging AB や log4net などの利用するロギングパッケージに応じて本機能を拡張する場合、以下のクラスを作成する。

- ILog 実装クラス

  ILog に定義されているログ出力用メソッドと、出力ログレベル確認用プロパティを実装する。個別のロギングパッケージの機能を実現するために、必要に応じてメソッドやプロパティを追加する。

- LogFactory 継承クラス

  ILog 実装クラスのインスタンスを取得するための抽象メソッドを実装する。

```
protected abstract ILog GetInstance(string className, params object[] args);
protected abstract ILog GetInstance(Type classType, params object[] args);
```

- LogFactory 継承クラスの指定

表 10 構成ファイル

ノード	属性	必須	値
/configuration/appSettings /add	key	○	構成要素名。 固定値。以下の値とする。 LogFactoryTypeName
			複数可
	value		LogFactory 継承クラスの完全修飾型名

LogFactory クラスは GetLogger メソッドでロガー(ILog 実装クラスのインターフェイス)を取得する時に、ログ出力に使用するロギングパッケージを決定する。使用するロギングパッケージに対応する LogFactory 継承クラスを、構成ファイルに記述する。

appSettings 要素内に add 要素を追加し、key 属性に”LogFactoryTypeName”、value 属性に LogFactory 継承クラスの完全修飾型名を設定する。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <!--使用するLogFactory継承クラスを指定する-->
    <add key="LogFactoryTypeName"
         value="SampleAP.Logging.SampleLog.SampleLogFactory, SampleAP" />
  </appSettings>
</configuration>
```

LogFactory 継承クラスの完全修飾型名

リスト 5 構成ファイルの設定例

CM-04 ビジネスロジック生成機能

■ 概要

ビジネスロジック設定ファイルの定義をもとに、ビジネスロジック名に対応するビジネスロジッククラスのインスタンスを生成する機能を提供する。

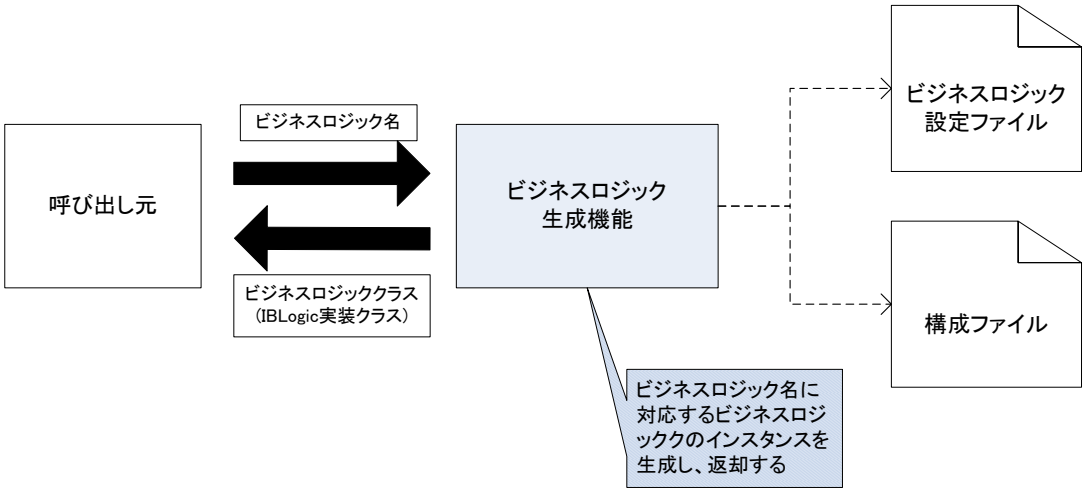


図 1 ビジネスロジック生成機能の概念図

全てのビジネスロジックは、IBLogic インタフェースを実装する必要がある。

ビジネスロジック設定ファイルを変更することで、再ビルドを行わずに利用するビジネスロジッククラスを変更することができる。

■ 使用方法

◆ 構成ファイル

構成ファイルに、blogicConfiguration 要素を有効化するためのクラスの設定と、ビジネスロジック設定ファイルのパスを指定する。ビジネスロジック設定ファイルは複数指定できる。

```
<configuration>
  <configSections>
    <section name="blogicConfiguration"
             type="TERASOLUNA.Fw.Common.Configuration.BLogic.BLogicConfigurationSection,
                   TERASOLUNA.Fw.Common"/>
  </configSections>

  <blogicConfiguration>
    <files>
      <!-- ビジネスロジック定義ファイルのパス設定 -->
      <file path="Config¥BLogicConfiguration01.config"/>
      <file path="Config¥BLogicConfiguration02.config"/>
    </files>
  </blogicConfiguration>
```

構成ファイルの blogicConfiguration 要素を有効化するためのクラスを定義する。

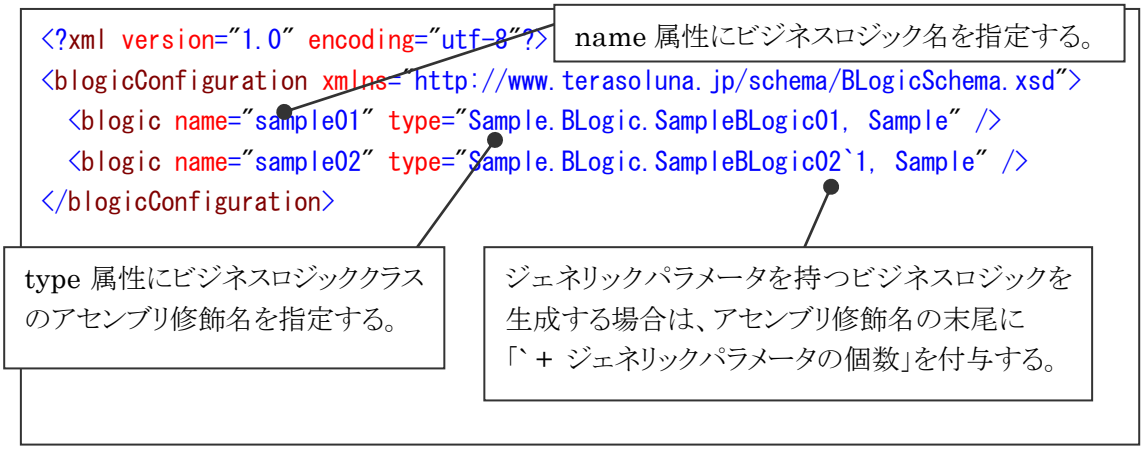
ビジネスロジック定義ファイルは複数設定が可能。

リスト 1 構成ファイルの記述例

◆ ビジネスロジック設定ファイル

ビジネスロジック設定ファイルにビジネスロジックのビジネスロジック名とアセンブリ修飾名を記述する。指定するビジネスロジック名は、全てのビジネスロジック設定ファイル内で一意でなければならない。

以下に、ビジネスロジック設定ファイルの記述例を示す。



リスト 2 ビジネスロジック設定ファイルの記述例

◆ 実装方法

(1) ビジネスロジックの作成

ジェネリックパラメータを持たないビジネスロジックと、ジェネリックパラメータを持つビジネスロジックの実装方法を以下に記す。

- ジェネリックパラメータを持たないビジネスロジックの実装

```
/// <summary>
/// サンプルBLogic。
/// </summary>
public class SampleBLogic01 : IBLogic
{
    public BLogicResult Execute(BLogicParam param)
    {
        BLogicResult blogicResult = new BLogicResult();
        blogicResult.ResultString = BLogicResult.SUCCESS;

        return blogicResult;
    }
}
```

リスト 3 ジェネリックパラメータを持たないビジネスロジックの実装例

- ジェネリックパラメータを持つビジネスロジックの実装

```
/// <summary>
/// サンプルBLogic（ジェネリックパラメータ付き）。
/// </summary>
public class SampleBLogic02<T> : IBLogic where T : DataSet, new()
{
    public BLogicResult Execute(BLogicParam param)
    {
        BLogicResult blogicResult = new BLogicResult();

        T resultDataSet = new T();
        resultDataSet.Tables[0].Rows.Add("VALUE01", "VALUE02");

        blogicResult.ResultString = BLogicResult.SUCCESS;
        blogicResult.ResultData = resultDataSet;

        return blogicResult;
}
```

リスト 4 ジェネリックパラメータを持つビジネスロジックの実装例

- BLogicParam のプロパティ

BLogicParam のプロパティ一覧を示す。

表 1 BLogicParam のプロパティ

項番	プロパティ名	説明
1	ParamData	ビジネスロジックの入力データセット。
2	Items	ビジネスロジックの入力パラメータを格納する IDictionary。

- BLogicResult のプロパティ

BLogicResult のプロパティ一覧を示す。

表 2 BLogicResult のプロパティ

項番	プロパティ名	説明
1	ResultString	ビジネスロジックの実行結果を表す文字列が格納される。ビジネスロジックが正常に実行された場合は、”success”（BLogicResult.SUCCESS）を設定すること。
2	ResultData	ビジネスロジックの実行結果を格納するデータセット。
3	Errors	ビジネスロジック内部で発生したエラーを格納するリスト。
4	Items	ビジネスロジックの実行結果を格納する IDictionary。

(2) ビジネスロジック生成クラスの実行

BLogicFactory の CreateBLogic メソッドを実行することで、ビジネスロジック設定ファイルに定義したビジネスロジッククラスのインスタンスを生成することができる。CreateBLogic メソッドの引数には、ビジネスロジック設定ファイルに定義したビジネスロジック名を指定すること。ジェネリックパラメータを持つビジネスロジックを生成する場合は、CreateBLogic メソッドの第二引数にジェネリックパラメータの型を設定する必要がある。

ジェネリックパラメータを持たないビジネスロジックを生成する場合と、ジェネリックパラメータを持つビジネスロジックを生成する場合の実装方法を以下に記す。

- ジェネリックパラメータを持たないビジネスロジックを生成する

```
// ビジネスロジックの作成
IBLogic blogic = BLogicFactory.CreateBLogic("sample01");

// ビジネスロジック入力クラスの作成
BLogicParam param = new BLogicParam();

// ビジネスロジック入力クラスへのパラーメータ設定

// ビジネスロジックの実行
BLogicResult result = blogic.Execute(param);

// ビジネスロジック実行結果の確認
if (BLogicResult.SUCCESS.Equals(result.ResultString))
{
    Console.Write("ビジネスロジック実行に成功しました。");
}
else
{
    Console.Write("ビジネスロジック実行に失敗しました。");
}
```

ビジネスロジック設定ファイルに定義されたビジネスロジック名を指定する。

リスト 5 ジェネリックパラメータを持たないビジネスロジックの生成例

- ジェネリックパラメータを持つビジネスロジックを生成する

```
// ビジネスロジックの作成
IBLogic blogic = BLogicFactory.CreateBLogic("sample02",
typeof(SampleDataSet).AssemblyQualifiedName);

// ビジネスロジック入力クラスの作成
BLogicParam param  = new BLogicParam();

// ビジネスロジック入力クラスへのパラーメータ設定

// ビジネスロジックの実行
BLogicResult result = blogic.Execute(param);

// ビジネスロジック実行結果の確認
if (BLogicResult.SUCCESS.Equals(result.ResultString))
{
    Console.Write("ビジネスロジック実行に成功しました。");
}
else
{
    Console.Write("ビジネスロジック実行に失敗しました。");
}
```

ビジネスロジック設定ファイルに定義されたビジネスロジック名を指定する。

ジェネリックパラメータのアセンブリ修飾名を指定する。

リスト 6 ジェネリックパラメータを持つビジネスロジックの生成例

- NopBLogic

BLogicFactory の CreateBLogic メソッドの引数に、空文字列もしくは null を設定した場合、BLogicFactory は NopBLogic のインスタンスを返却する。NopBLogic は BLogicResult の ResultString に”success”を設定する処理のみを行う。

■ 拡張ポイント

◆ ビジネスロジック生成クラスの差し替え

構成ファイルの AppSettings セクションに BLogicFactory を拡張したクラスのアセンブリ修飾名を指定することで、利用するビジネスロジック生成クラスの実装を差し替えることができる。アセンブリ修飾名の指定を省略した場合は、TERASOLUNA が提供する BLogicFactory クラスが使用される。なお、アセンブリ修飾名を指定する AppSettings セクションのキーは「BLogicFactoryTypeName」である。

以下に、ビジネスロジック生成クラスの差し替えを設定する例を示す。

```
<configuration>
  <appSettings>
    <add key="BLogicFactoryTypeName"
         value="Sample.BLogic.BLogicFactoryEx, Sample "/>
  </appSettings>
</configuration>
```

リスト 7 ビジネスロジック生成クラスを差し替える設定例

WA-01 画面遷移管理機能

■ 概要

本機能は、ページ設定ファイルに定義した画面遷移情報に基づいて画面遷移を行う機能である。

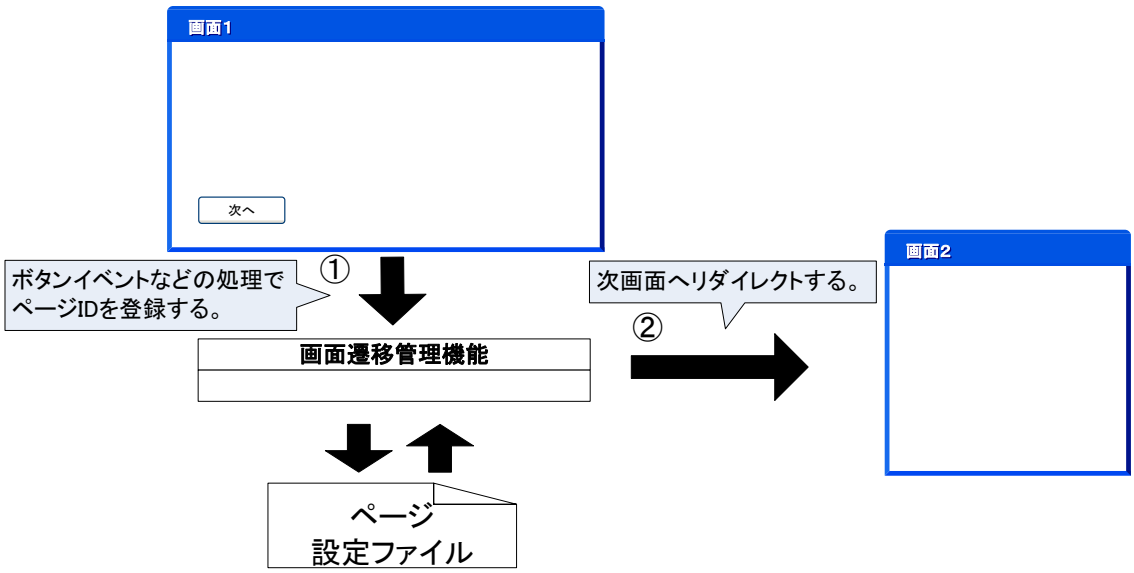


図 1 画面遷移管理機能

ページ設定ファイルには、画面に対して一意なページ ID と画面の URL を定義する。ボタンクリックイベントなどで遷移先ページ ID を指定することで、ページ設定ファイルから遷移先ページの URL を取得し、次画面へリダイレクトする。遷移先 URL をページ設定ファイルから取得するため、再ビルドを行わずに画面遷移先を変更することができる。また、ページ設定ファイルは、Web 構成ファイル(web.config)に複数定義できるため、ユースケースごとにページ設定ファイルを分割できる。

■ 使用方法

◆ Web 構成ファイル

本機能を有効にする場合、Web 構成ファイル(web.config)に HttpModule およびページ設定ファイルのパスを定義する。

表 1 Web 構成ファイル

ノード	属性	必須	値
/configuration/configSections/section			
	name	○	構成要素名。 固定値、以下を指定する。 pageConfiguration
	type	○	構成設定の処理を行う構成セクションハンドラクラス名。 固定値、以下を指定する。 TERASOLUNA.Fw.Web.Configuration.Page.PageConfigurationSection,TERASOLUNA.Fw.Web
/configuration/pageConfiguration/files/file			複数可
	path	○	ページ設定ファイルの保存先のパスを指定する。
/System.web/httpModules	name	○	モジュール登録名。 固定値、以下を指定する。 TransitionListenerImpl
	type	○	クラス名と、クラスが含まれるアセンブリ名。 固定値、以下を指定する。 TERASOLUNA.Fw.Web.HttpModule.TransitionListenerImpl, TERASOLUNA.Fw.Web

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  …
  <configSections>
    <section name="pageConfiguration"
                type="TERASOLUNA.Fw.Web.Configuration.Page.PageConfigurationSection,
                TERASOLUNA.Fw.Web"/>
  </configSections>
  <pageConfiguration>
    <files>
      <file path="Config¥PageConfiguration01.config" />
      <file path="Config¥PageConfiguration02.config" />
    </files>
  </pageConfiguration>
  <system.web>
    <httpModules>
      <add name="TransitionListenerImpl"
            type="TERASOLUNA.Fw.Web.HttpModule.TransitionListenerImpl,
                  TERASOLUNA.Fw.Web" />
    </httpModules >
  </system.web>
  …
</configuration>
```

リスト 1 Web 構成ファイル記述例

◆ ページ設定ファイル

ページ設定ファイルには、ページ毎のページ ID と仮想アプリケーションルートパス以降のパスを関連付ける。ページ設定ファイルについては、画面遷移保証機能、二重押下防止機能、エラー画面遷移機能と共有することができる。

表 2 ページ設定ファイル

<table>
<tr><th>ノード</th><th>属性</th><th>必須</th><th>値</th></tr>
<tr><td>/pageConfiguration</td><td>xmlns</td><td>○</td><td>Namespace 名。
固定値、以下を指定する。
http://www.terasoluna.jp/schema/PageSchema.xsd</td></tr>
<tr><td rowspan="6">/pageConfiguration/page</td><td></td><td></td><td>複数可。</td></tr>
<tr><td>name</td><td>○</td><td>ページ ID。重複不可。</td></tr>
<tr><td>path</td><td>○</td><td>ページパス。重複不可。</td></tr>
<tr><td>preventDoubleSubmit</td><td></td><td>二重押下防止フラグ。
デフォルト値は、on。 on、または off を指定可能。
<table>
<tr><th>設定値</th><th>説明</th></tr>
<tr><td>on</td><td>ページに対して二重押下防止機能を有効にする。</td></tr>
<tr><td>off</td><td>ページに対して二重押下防止機能を無効にする。</td></tr>
</table></td></tr>
<tr><td>checkToken</td><td></td><td>トークンチェックフラグ。
デフォルト値は、on。on、または off を指定可能。
<table>
<tr><th>設定値</th><th>説明</th></tr>
<tr><td>on</td><td>ページに対してトークンチェック機能を有効にする。</td></tr>
<tr><td>off</td><td>ページに対してトークンチェック機能を無効にする。</td></tr>
</table></td></tr>
<tr><td>updateToken</td><td></td><td>トークン更新フラグ。
デフォルト値は、on。on、または off を指定可能。
<table>
<tr><th>設定値</th><th>説明</th></tr>
<tr><td>on</td><td>ページ遷移時にトークンを更新する。</td></tr>
<tr><td>off</td><td>ページ遷移時にトークンを更新しない。</td></tr>
</table></td></tr>
</table>

ページ ID とパスは、すべてのページ設定ファイルにおいて一意である必要がある。異なるページ設定ファイルに重複するページ ID やパスがある場合、設定ファイルの読み込み時に TERASOLUNA フレームワークから ConfigurationErrorsException がスローされる。

preventDoubleSubmit 属性は「二重押下防止機能」、checkToken 属性、updateToken 属

性については「画面遷移保証機能」を参照のこと。

```
<?xml version="1.0" encoding="utf-8" ?>
<pageConfiguration xmlns="http://www.terasoluna.jp/schema/PageSchema.xsd">
    <page name="n01" path="/UI/n01.aspx"/>
    <page name="n02" path="/UI/n02.aspx"/>
    <page name="m01" path="/UI/t1/n01.aspx"/>
    <page name="m02" path="/UI/t1/n02.aspx"/>
    <page name="m03" path="/UI/t1/n03.aspx"/>
    <page name="m04" path="/UI/t2/n01.aspx"/>
    <page name="m05" path="/UI/t2/n02.aspx"/>
    <page name="m06" path="/UI/t2/n03.aspx"/>
</pageConfiguration>
```

リスト 2 ページ設定ファイル記述例

◆ 実装方法

WebUtils クラスの Transit メソッドにページ ID やクエリ文字列を指定する。

```
///  "http://<ホスト名>/<アプリケーションルート名>/UI/n01.aspx"
///  へ遷移するクリックイベント実装例
protected void Button1_Click(object sender, EventArgs e)
{
    WebUtils.Transit("n01");
}

///  "http://<ホスト名>/<アプリケーションルート名>/UI/n01.aspx?k01=v01&k02=v02"
///  へ遷移するクリックイベント実装例
protected void Button2_Click(object sender, EventArgs e)
{
     IDictionary<string, string> querys = new Dictionary<string, string>();
     querys.Add("k01", "v01");
     querys.Add("k02", "v02");
     WebUtils.Transit("n01", querys);
}

///  "http://<ホスト名>/<アプリケーションルート名>/UI/t1/n01.aspx?k03=v03"
///  へ遷移するクリックイベント実装例
protected void Button3_Click(object sender, EventArgs e)
{
      WebUtils.Transit("m01", "k03", "v03");
}
```

リスト 3 実装例

■ 関連機能

- WA-02 画面遷移保証機能
- WA-03 二重押下防止機能
- WA-04 エラー画面遷移機能

WA-02 画面遷移保証機能

■ 概要

本機能は、ページ設定ファイルに定義したトークン情報に基づいて画面への不正アクセスを防止する機能である。

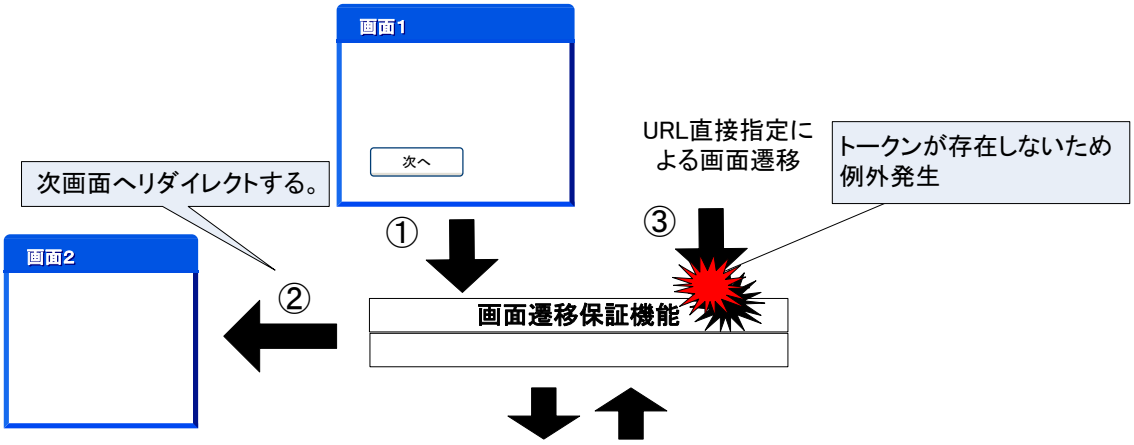


図 1 画面遷移保証機能

ページ設定ファイルには、画面に対して一意なページ ID、画面の URL、トークンチェックフラグ、トークン更新フラグを定義する。遷移先ページへのリダイレクトや URL 直接指定による画面遷移時に、遷移先ページのトークンチェックフラグが有効であれば、不正な画面遷移ではないか確認する。もし、不正な画面遷移である場合は、例外が発生する。また、ページ設定ファイルは、Web 構成ファイル(web.config)に複数定義できるため、ユースケースごとにページ設定ファイルを分割できる。

■ 使用方法

◆ Web 構成ファイル

本機能を有効にする場合、Web 構成ファイル(web.config)に HttpModule とページ設定ファイルを定義する。

表 1 Web 構成ファイル

ノード	属性	必須	値
/configuration/configSections/section			
	name	○	構成要素名。 固定値。以下の値を指定する。 pageConfiguration
	type	○	構成設定の処理を行う構成セクションハンドラクラス名。 固定値。以下の値を指定する。 TERASOLUNA.Fw.Web.Configuration.Page.PageConfigurationSection, TERASOLUNA.Fw.Web
/configuration/pageConfiguration/files/file			複数可
	path	○	ページ設定ファイルのアプリケーションルートからのパス。
/System.web/httpModules/add			複数可
	name	○	Http モジュール登録名。 固定値。以下の値を指定する。 TokenProcessorImpl
	type	○	クラス名とクラスが含まれる、アセンブリ名。 固定値。以下の値を指定する。 TERASOLUNA.Fw.Web.HttpModule.TokenProcessorImpl, TERASOLUNA.Fw.Web

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  …
  <configSections>
    <section name="pageConfiguration"
                  type="TERASOLUNA.Fw.Web.Configuration.Page.PageConfigurationSection,
                  TERASOLUNA.Fw.Web"/>
  </configSections>
  <pageConfiguration>
    <files>
      <file path="Config¥PageConfiguration01.config" />
      <file path="Config¥PageConfiguration02.config" />
    </files>
  </pageConfiguration>
  <system.web>
    <httpModules>
      <add name="TokenProcessorImpl"
                type="TERASOLUNA.Fw.Web.HttpModule.TokenProcessorImpl,
                      TERASOLUNA.Fw.Web"/>
    </httpModules>
  </system.web>
  …
```

リスト 1 Web 構成ファイル記述例

◆ ページ設定ファイル

ページ設定ファイルにより、各ページのページ ID と仮想アプリケーションルートパス以降のパスを関連付ける。ページ設定ファイルについては、画面遷移管理機能、二重押下防止機能、エラー画面遷移機能と共有することができる。

表 2 ページ設定ファイル

ノード	属性	必須	値
/pageConfiguration	xmlns	○	Namespace 名。 固定値、以下を指定する。 http://www.terasoluna.jp/schema/PageSchema.xsd
/pageConfiguration/page			複数可。
	name	○	ページ ID。重複不可。
	path	○	ページパス。重複不可。
	preventDoubleSubmit		二重押下防止フラグ。 デフォルト値は、on。on、または off を指定可能。 設定値 説明 on ページに対して二重押下防止機能を有効にする。 off ページに対して二重押下防止機能を無効にする。
	checkToken		トークンチェックフラグ。 デフォルト値は、on。on、または off を指定可能。 設定値 説明 on ページに対してトークンチェック機能を有効にする。 off ページに対してトークンチェック機能を無効にする。
	updateToken		トークン更新フラグ。 デフォルト値は、on。on、または off を指定可能。 設定値 説明 on ページ遷移時にトークンを更新する。 off ページ遷移時にトークンを更新しない。

【checkToken 属性について】

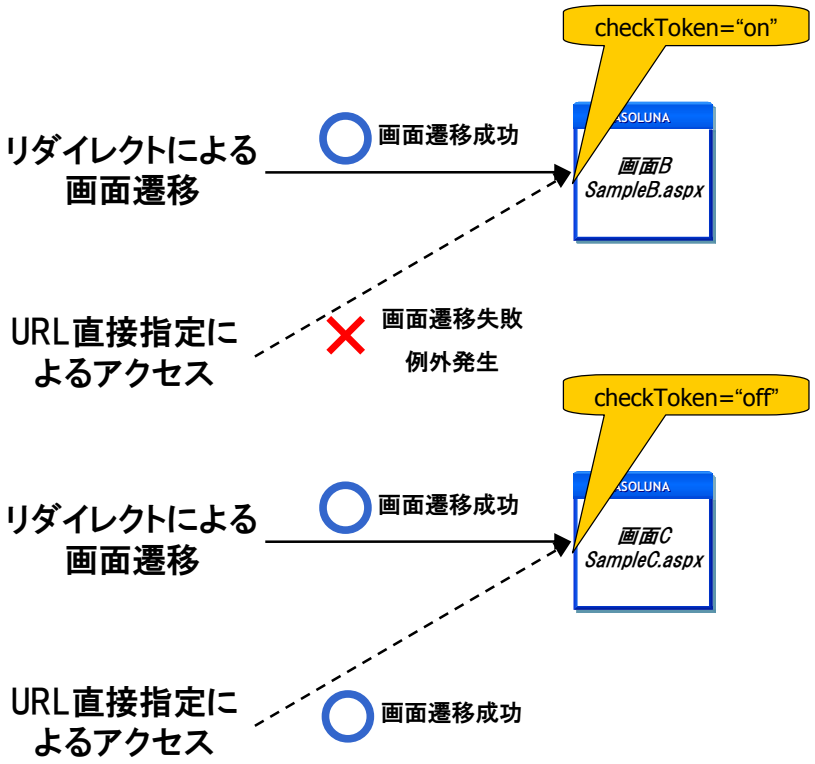


図 2 checkToken の動作例

checkToken 属性は、指定されたページに対してトークンをチェックするかどうかを示すフラグである。二重送信を防止したいページや、URL を直接指定してのアクセスを防止したいページにはトークンチェックを有効(checkToken="on")にする。遷移先画面のトークンチェックを有効にすることで、Web アプリケーション内から遷移先画面へリダイレクトする時に、画面遷移保証機能が自動でトークンの設定と確認をする。

【updateToken 属性について】

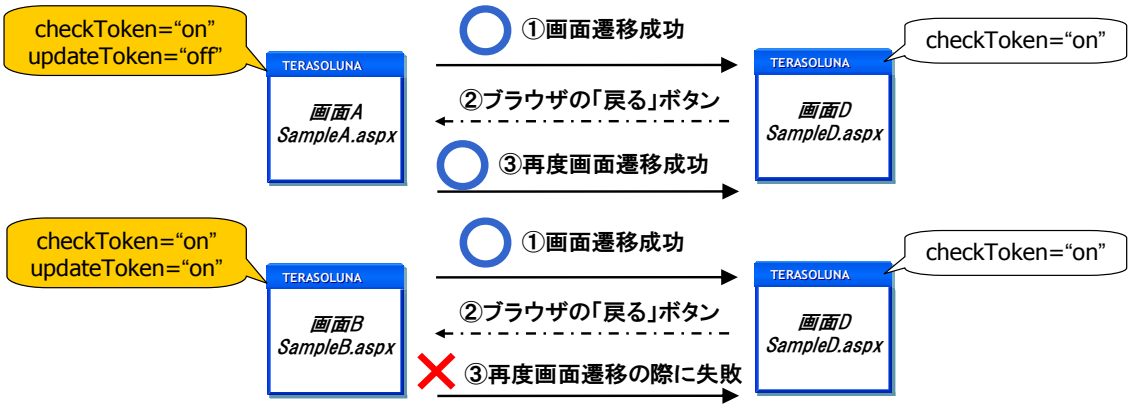


図 3 updateToken の動作例

updateToken 属性は、トークンを更新するかどうかを示すフラグである。ブラウザの「戻る」ボタンで戻ったページから再度 checkToken=”on”の画面への画面遷移を許可するためには、トークン更新を無効(updateToken=”off”)にする。

```
<?xml version="1.0" encoding="utf-8" ?>
<pageConfiguration xmlns="http://www.terasoluna.jp/schema/PageSchema.xsd">
    <page name="n01" path="/SampleA.aspx”
                        checkToken="on" updateToken=”off”/>
    <page name="n02" path="/SampleB.aspx"checkToken="on" updateToken=”on”/>
    <page name="n03" path="/SampleC.aspx" checkToken="off"/>
    <page name="n04" path="/SampleD.aspx" checkToken=”on”/>
</pageConfiguration>
```

リスト 2 ページ設定ファイル設定例

■ 関連機能

- WA-01 画面遷移管理機能
- WA-03 二重押下防止機能
- WA-04 エラー画面遷移機能

WA-03 二重押下防止機能

■ 概要

本機能は、ボタンが押下された際に画面内の全てのボタンを無効(disable)にして、リクエストの二重送信を防止する機能である。

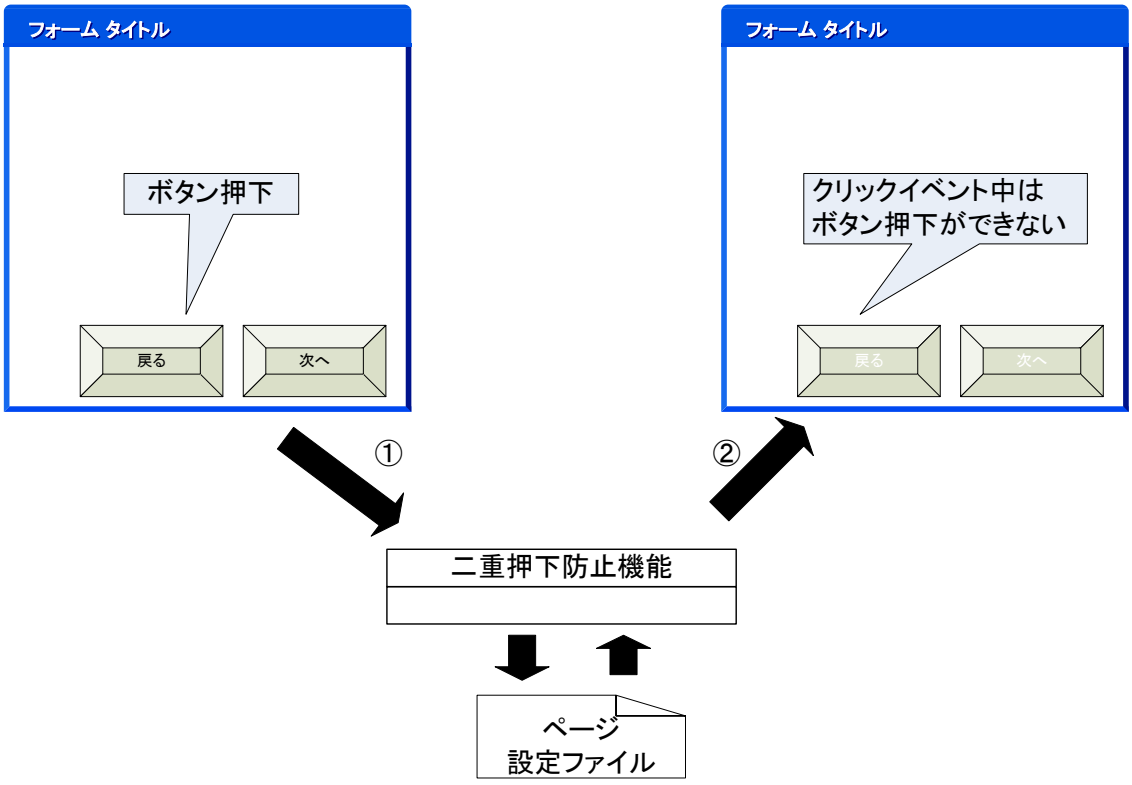


図 1 二重押下防止機能

ページ設定ファイルには、画面に対して一意なページ ID、画面の URL、二重押下防止フラグを定義する。二重押下防止フラグが有効になっている画面のボタンをクリックした場合、画面内の全てのボタンを無効(disable)にする。ページ設定ファイルは、Web 構成ファイル(web.config)に複数定義できるため、ユースケースごとにページ設定ファイルを分割できる。

■ 使用方法

◆ Web 構成ファイル

本機能を有効にする場合、Web 構成ファイル(web.config)にページ設定ファイルパスおよびHttpModule を定義する。

表 1 Web 構成ファイル

ノード	属性	必須	値
/configuration/configSections/section	-	-	複数可
	name	-	構成要素名。 固定値、以下の値とする pageConfiguration
	type	-	構成設定の処理を行う構成セクションハンドラクラス名。 固定値、以下の値とする。 TERASOLUNA.Fw.Web.Configuration.Page.PageConfigurationSection, TERASOLUNA.Fw.Web
/configuration/pageConfiguration/files/file	-	-	複数可
	path	○	ページ設定ファイルの保存先のパスを指定する。
/configuration/System.web/httpModules	name	○	モジュール登録名。 固定値、以下の値とする。 PreventDoubleSubmitImpl
	type	○	クラス名と、クラスが含まれるアセンブリ名。 固定値、以下の値とする。 TERASOLUNA.Fw.Web.HttpModule.PreventDoubleSubmitImpl, TERASOLUNA.Fw.Web

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  …
  <configSections>
    <section name="pageConfiguration"
                  type="TERASOLUNA.Fw.Web.Configuration.Page.PageConfigurationSection,
                  TERASOLUNA.Fw.Web"/>
  </configSections>
  <pageConfiguration>
    <files>
      <file path="Config¥PageConfiguration.config" />
    </files>
  </pageConfiguration>
  …
  <system.web>
    <httpModules>
      <add name="PreventDoubleSubmitImpl"
                type="TERASOLUNA.Fw.Web.HttpModule.PreventDoubleSubmitIm
                pl, TERASOLUNA.Fw.Web" />
    </ httpModules >
  </ system.web>
  …
```

リスト 1 Web 構成ファイル記述例

◆ ページ設定ファイル

ページ設定ファイルにより、各ページのページ ID、仮想アプリケーションルートパス以降のパス、二重押下防止フラグを関連付ける。ページ設定ファイルについては、画面遷移管理機能、画面遷移保証機能、エラー画面遷移機能と共有することができる。

表 2 ページ設定ファイル

ノード	属性	必須	値
/pageConfiguration	xmlns	○	Namespace 名。 固定値、以下を指定する。 http://www.terasoluna.jp/schema/PageSchema.xsd
/pageConfiguration/page	-	-	複数可。
	name	○	ページ ID。重複不可。
	path	○	ページパス。重複不可。
	preventDoubleSubmit	-	二重押下防止フラグ。 デフォルト値は、on。on、または off を指定可能。 設定値　説明 on　ページに対して二重押下防止機能を有効にする。 off　ページに対して二重押下防止機能を無効にする。
	checkToken	-	トークンチェックフラグ。 デフォルト値は、on。on、または off を指定可能。 設定値　説明 on　ページに対してトークンチェック機能を有効にする。 off　ページに対してトークンチェック機能を無効にする。
	updateToken	-	トークン更新フラグ。 デフォルト値は、on。on、または off を指定可能。 設定値　説明 on　ページ遷移時にトークンを更新する。 off　ページ遷移時にトークンを更新しない。

```
<?xml version="1.0" encoding="utf-8" ?>
<pageConfiguration   xmlns="http://www.terasoluna.jp/schema/PageSchema.xsd">
  <page name="menu" path="/UI/menu.aspx" preventDoubleSubmit="on" />
</pageConfiguration>
```

リスト 2 ページ設定ファイル記述例

■ 関連機能

- WA-01 画面遷移管理機能
- WA-02 画面遷移保証機能
- WA-04 エラー画面遷移機能

WA-04 エラー画面遷移機能

■ 概要

本機能は、Web アプリケーションで例外が発生した時に、エラー画面遷移設定ファイル、ページ設定ファイルに定義した情報に基づいて画面遷移を行う機能である。

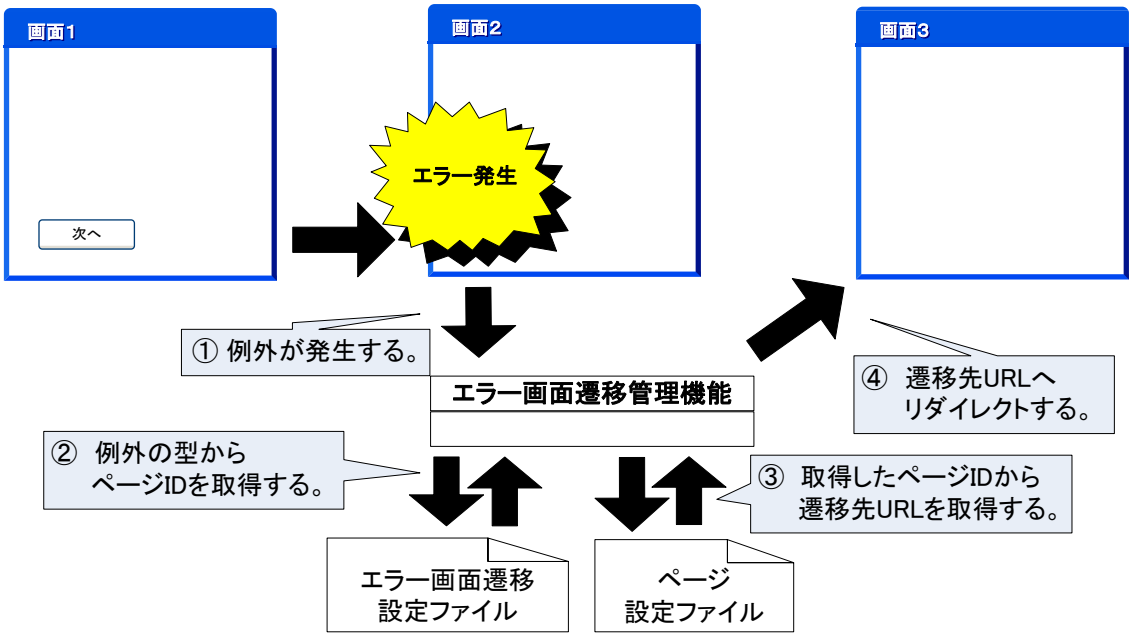


図 1 エラー画面遷移管理機能

エラー画面遷移設定ファイルには、例外クラスの完全修飾名とページ ID を定義する。ページ設定ファイルには、ページ ID と画面の URL を定義する。画面遷移やポストバック実行時に例外が発生した場合、発生した例外クラスに対応したページ ID をエラー画面遷移設定ファイルから取得する。取得したページ ID に対応した遷移先の URL をページ設定ファイルから取得し、エラー画面へリダイレクトする。エラー画面遷移設定ファイル及びページ設定ファイルは、Web 構成ファイル(web.config)に複数定義できるため、ユースケースごとに分割できる。

■ 使用方法

エラー画面遷移機能を使う上で、必要な Web 構成ファイル、ページ設定ファイル、エラー画面遷移設定ファイル、IIS の設定について解説する。

◆ Web 構成ファイル

本機能を有効にする場合、Web 構成ファイル(web.config)に HttpModule、ページ設定ファイルパス、エラー画面遷移設定ファイルパスを定義する。

表 1 Web 構成ファイル

ノード	属性	必須	値
/configuration/configSections/section	-	-	複数可
	name	-	構成要素名。 固定値、以下の値とする。 ・pageConfiguration ・exceptionTransitionConfiguration
	type	-	構成設定の処理を行う構成セクション ハンドラ クラス名。 固定値、以下の値とする。 ・TERASOLUNA.Fw.Web.Configuration.Page.PageConfigurationSection, TERASOLUNA.Fw.Web ・TERASOLUNA.Fw.Web.Configuration.ExceptionTransition.ExceptionTransitionConfigurationSection, TERASOLUNA.Fw.Web
/configuration/pageConfiguration/files/file	-	-	複数可
	path	○	ページ設定ファイル。
/configuration/ExceptiontransitionConfiguration	mode	-	エラー画面遷移フラグ。 デフォルト値は、on。 設定値 説明 on エラー画面遷移機能を有効にする。 off エラー画面遷移機能を無効にする。
	logging	-	エラー画面遷移ログ出力フラグ。 デフォルト値は、off。 IIS ログに例外情報を出力するためには、エラー画面遷移ログ出力フラグを on にする以外にも、IIS でログを出力するための設定が必要である。本機能説明書の「IIS の設定」を参照すること。 設定値 説明 on IIS ログを出力する。 off IIS ログを出力しない
/configuration/ExceptiontransitionConfiguration/files/file			複数可
	path	○	エラー画面遷移設定ファイル。
/System.web/httpModules	name	○	モジュール登録名。 固定値、以下を指定する。 ExceptionTransitionListenerImpl
	type	○	クラス名と、クラスが含まれるアセンブリ名。 固定値、以下を指定する。 TERASOLUNA.Fw.Web.HttpModule.ExceptionTransitionListenerImpl, TERASOLUNA.Fw.Web

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  …
  <configSections>
    <section name="pageConfiguration"
                type="TERASOLUNA.Fw.Web.Configuration.Page.PageConfigurationSection,
                TERASOLUNA.Fw.Web"/>
    <section name="exceptionTransitionConfiguration"
                type="TERASOLUNA.Fw.Web.Configuration.ExceptionTransition.ExceptionTra
                nsitionConfigurationSection, TERASOLUNA.Fw.Web"/>
  </configSections>
  <pageConfiguration>
    <files>
      <file path="Config¥PageConfiguration.config" />
    </files>
  </pageConfiguration>
  <exceptionTransitionConfiguration mode="on" logging="on">
    <files>
      <file path="Config¥ExceptionTransition.config"/>
    </files>
  </exceptionTransitionConfiguration>
  <customErrors mode="On" defaultRedirect="DefaultErrorPage.aspx">
  </customErrors>
   <system.web>
     <httpModules>
       <add name="ExceptionTransitionListenerImpl"
       type="TERASOLUNA.Fw.Web.HttpModule.ExceptionTransitionListenerImpl,
   TERASOLUNA.Fw.Web" />
     </httpModules >
  </system.web>
 …
```

リスト 1 Web 構成ファイル記述例

◆ ページ設定ファイル

ページ設定ファイルにより、各ページの ID と仮想アプリケーションルートパス以降のパスを関連付ける。ページ設定ファイルについては、画面遷移管理機能、画面遷移保証機能、二重押下防止機能と共有することができる。

表 2 ページ設定ファイル

ノード	属性	必須	値
/pageConfiguration	xmlns	○	Namespace 名。 固定値、以下を指定する。 http://www.terasoluna.jp/schema/PageSchema.xsd
/pageConfiguration/page	-	-	複数可。
	name	○	ページ ID。重複不可。
	path	○	ページパス。重複不可。
	preventDoubleSubmit	-	二重押下防止フラグ。 デフォルト値は、on。on または off を指定可能。 設定値 説明 on ページに対して二重押下防止機能を有効にする。 off ページに対して二重押下防止機能を無効にする。
	checkToken	-	トークンチェックフラグ。 デフォルト値は、on。on または off を指定可能。 設定値 説明 on ページに対してトークンチェック機能を有効にする。 off ページに対してトークンチェック機能を無効にする。
	updateToken	-	トークン更新フラグ。 デフォルト値は、on。on または off を指定可能。 設定値 説明 on ページ遷移時にトークンを更新する。 off ページ遷移時にトークンを更新しない。

```
<?xml version="1.0" encoding="utf-8" ?>
<pageConfiguration xmlns="http://www.terasoluna.jp/schema/PageSchema.xsd">
  <page name="ExTranMain" path="/ExceptionTransitionListerImpl/Main.aspx"/>
  <page name="ExTranError"
                path="/ExceptionTransitionListerImpl/ExTranErrorPage.aspx" />
  <page name="CustomError"
              path="/ExceptionTransitionListerImpl/CustomErrorPage.aspx" />
</pageConfiguration>
```

リスト 2 ページ設定ファイル記述例

◆ エラー画面遷移設定ファイル

エラー画面遷移設定ファイルにより、例外クラスの完全修飾名とページ ID を関連付ける。

表 3 エラー画面遷移設定ファイル

ノード	属性	必須	値
/ exceptionTransitionConfiguration	xmlns	○	Namespace 名。 固定値、以下を指定する。 http://www.terasoluna.jp/schema/ExceptionTransitionSchema.xsd
/exceptionTransitionConfiguration /exceptionTransition			複数可
	exceptionType	○	例外クラスの完全修飾名。
	nextPage	○	遷移先ページ ID。ページ設定ファイルで指定したページ ID を指定する。

```
<?xml version="1.0" encoding="utf-8" ?>
<exceptionTransitionConfiguration
xmlns="http://www.terasoluna.jp/schema/ExceptionTransitionSchema.xsd">
  <exceptionTransition
       exceptionType="System.ArgumentException" nextPage="ExTranError" />
  <exceptionTransition
       exceptionType="TERASOLUNA.Fw.Web.Controller.InvalidRequestException"
       nextPage="CustomError" />
</exceptionTransitionConfiguration>
```

リスト 3 エラー画面遷移管理ファイル記述例

◆ IIS の設定

IIS にログを出力するためには、Web 構成ファイルでのエラー画面遷移ログ出力フラグを有効(logging="on")にする以外に、IIS の設定が必要である。以下の設定を行うことで、IIS で例外ログを出力することができる。

- Web サーバの管理ツールで、拡張ログオプション"URI クエリ"にチェックを付ける。
  (下図は IIS 5 の設定画面だが、IIS 7.5 でも同様の箇所で設定をする。)

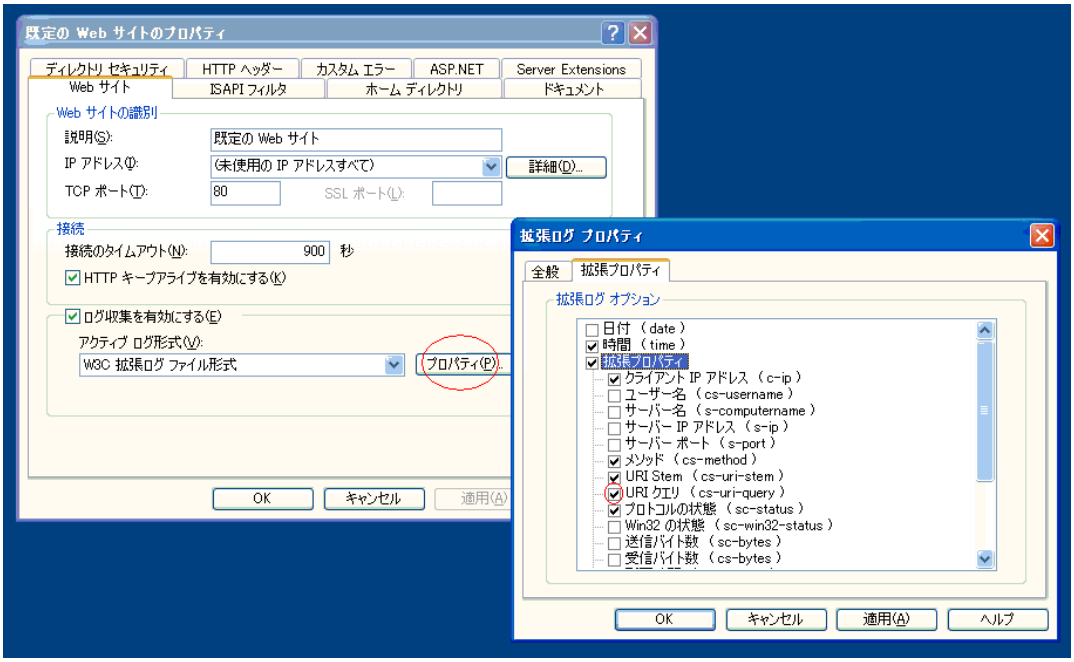


図 2 IIS 設定画面

```
#Software: Microsoft Internet Information Services 5.0
#Version: 1.0
#Date: 2006-03-29 04:40:52
#Fields: date time c-ip cs-method cs-uri-stem cs-uri-query sc-status sc-bytes cs-bytes time-taken
2006-03-29 04:40:52 127.0.0.1 GET
 /WebSI_AP01/ExceptionTransitionListerImpl/Main.aspx - 200 1334 330 5187
2006-03-29 04:40:54 127.0.0.1 POST
 /WebSI_AP01/ExceptionTransitionListerImpl/Main.aspx
+Source+:App_Web_mfiblpop;
+Message+:+値が有効な範囲にありません。;
+StackTace+:+++
+場所+Main.ButtonThrowArgumentEx_Click(Object+sender,+EventArgs+e)++++
+場所+System.Web.UI.WebControls.Button.OnClick(EventArgs+e)++++
 （中略）
 2006-03-29 04:40:54 127.0.0.1 GET
 /WebSI_AP01/ExceptionTransitionListerImpl/ExTranErrorPage.aspx
__Token=9673d8c671e644a4a28fd732ef637ac5 200 1523 646 120
```

リスト 4 ログ出力例

■ 関連機能

- WA-01 画面遷移管理機能

WB-01 リクエストコントローラ機能

■ 概要

本機能は、クライアントからのリクエスト要求に対応する入力値検証、ビジネスロジックを実行し、処理結果をクライアントへ送信する機能である。

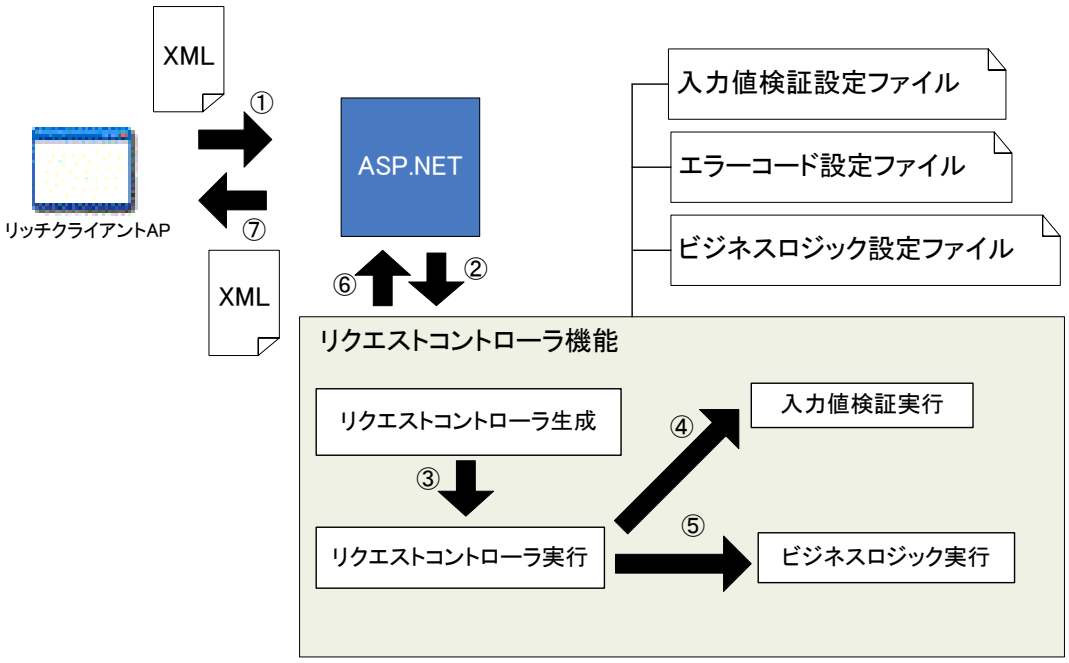


図 1 リクエストコントローラ機能

クライアントから送信されるリクエスト情報に基づいて、実行するリクエストコントローラが生成される。生成されたリクエストコントローラが入力値検証、ビジネスロジックを実行する。リクエストコントローラで例外が発生した場合、例外に対応したエラーコードをエラーコード設定ファイルから取得し、エラー電文(XML)にエラーコードを含めてクライアントへ送信する。

リクエストコントローラに関連する機能として、入力値検証は、TERASOLUNA で提供している入力値検証機能、ビジネスロジックは TERASOLUNA で提供しているビジネスロジック生成機能を利用する。入力値検証機能、ビジネスロジック機能の詳細な内容は、「CM-02 入力値検証機能」、「CM-04 ビジネスロジック生成機能」を参照すること。本機能で説明を行うリクエストコントローラ機能は、XML を送受信するリクエストコントローラである。クライアントがファイル送信する処理に対応するリクエストコントローラは、「WB-02 ファイルアップロード機能」、ファイル受信する処理に対応するリクエストコントローラは、「WB-03 ファイルダウンロード機能」を参照すること。

■ 使用方法

リクエストコントローラ機能を使う上で、必要な Web 構成ファイル、ビジネスロジック設定ファイル、エラーコード設定ファイル、入力値検証設定ファイル、ビジネスロジックの実装方法について解説する。

◆ Web 構成ファイル

本機能を有効にする場合、Web 構成ファイル(web.config)に IHttpHandlerFactory およびビジネスロジック設定ファイルのパスを定義する。

表 1 Web 構成ファイル

ノード	属性	必須	値
/configuration/configSections/section			複数可
	name	○	構成要素名。 固定値、以下を指定する。 ・blogicConfiguration ・exceptionCodeConfiguration
	type	○	構成設定の処理を行う構成セクションハンドラクラス名。以下の固定値を指定。 ・TERASOLUNA.Fw.Common.Configuration.BLogic.BLogicConfigurationSection, TERASOLUNA.Fw.Common ・TERASOLUNA.Fw.Web.Configuration.ExceptionCode.ExceptionCodeConfigurationSection, TERASOLUNA.Fw.Web
/configuration/blogicConfiguration/files/file			複数可
	path	○	ビジネスロジック設定ファイルパス。
/configuration/exceptionCodeConfiguration/files/file			複数可
	path	○	エラーコード設定ファイルパス。
/configuration/system.web/httpHandlers/add			複数可
	verb	○	受け付けるリクエストの HTTP メソッド。以下を指定する。 POST
	path	○	リクエストを受け付ける URL となるパス。アプリケーションルートからのパスを設定する。
	type	○	リクエストを受け付ける HTTP ハンドラクラスのアセンブリ修飾名。TERASOLUNA で提供しているリクエストコントローラをそのまま利用する場合は、以下を指定。 TERASOLUNA.Fw.Web.Controller.RequestControllerFactory, TERASOLUNA.Fw.Web

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="blogicConfiguration"
     type="TERASOLUNA.Fw.Common.Configuration.BLogic.BLogicConfigurationSection,
           TERASOLUNA.Fw.Common"/>
    <section name="exceptionCodeConfiguration"
     type="TERASOLUNA.Fw.Web.Configuration.ExceptionCode.ExceptionCodeConfigurati
     onSection, TERASOLUNA.Fw.Web"/>
  </configSections>
  <blogicConfiguration>
    <files>
      <file path="Config¥BLogicConfiguration.config"/>
    </files>
  </blogicConfiguration>
  <exceptionCodeConfiguration>
    <files>
      <file path="Config¥ErrorConfiguration.config"/>
    </files>
  </exceptionCodeConfiguration>
  <system.web>
    <httpHandlers>
      <add verb="POST" path="RequestController.aspx"
           type="TERASOLUNA.Fw.Web.Controller.RequestControllerFactory,
                 TERASOLUNA.Fw.Web"/>
    </httpHandlers>
  </system.web>
</configuration>
```

受け付けるリクエストのHTTP メソッドとして、POST を指定。

リクエストを受け付けるURL を指定。
(アプリケーションルートからの相対パス)

リスト 1 Web 構成ファイル記述例

◆ ビジネスロジック設定ファイル

ビジネスロジック設定ファイルに、IBLogic インタフェースを実装するクラスを設定する。

表 2 ビジネスロジック設定ファイル

ノード	属性	必須	値
/blogicConfiguration	xmlns	○	XML スキーマの名前空間。 以下の固定値を指定。 http://www.terasoluna.jp/schema/BLogicSchema.xsd
/blogicConfiguration/blogic			複数可
	name	○	ビジネスロジック名。重複不可。
	type	○	IBLogic インタフェースを実装したクラス名と、クラスが含まれるアセンブリ名。

```
<?xml version="1.0" encoding="utf-8" ?>
<blogicConfiguration xmlns="http://www.terasoluna.jp/schema/BLogicSchema.xsd">
  <blogic name="sampleA" type="SampleProject.Sample.SampleA, SampleProject" />
  <blogic name="sampleB" type="SampleProject.Sample.SampleB, SampleProject" />
</blogicConfiguration>
```

リスト 2 ビジネスロジック設定ファイル記述例

◆ エラーコード設定ファイル

エラーコード設定ファイルに、例外とエラーコードの対応を設定する。エラーコード設定ファイルに設定した例外クラスまたは例外クラスのサブクラスがリクエストコントローラで発生した場合、例外に対応したエラーコードがクライアントへ XML 形式で送信される。クライアントへ送信される内容は本機能説明書の内部構成を参照すること。

表 3 エラーコード設定ファイル

ノード	属性	必須	値
/exceptionConfiguration/exceptionCode	-	-	例外とエラーコードのマッピングを保持する要素 ※複数可
	exceptionType	○	エラーコードをマッピングする例外の完全クラス名。
	code	○	設定するエラーコード。

```
<?xml version="1.0" encoding="utf-8" ?>
<exceptionCodeConfiguration
  xmlns="http://www.terasoluna.jp/schema/ExceptionCodeSchema.xsd">
  <exceptionCode exceptionType="System.Exception" code="E000001" />
</exceptionCodeConfiguration>
```

リスト 3 エラーコード設定ファイル記述例

◆ 入力値検証設定ファイル

入力値検証設定ファイルの設定方法や入力値検証の動作内容については、機能説明書「CM-02 入力値検証機能」を参照すること。

◆ 実装方法

IBLogic インタフェースを実装し、実装したクラスのクラス属性にリクエストタイプ名、入力データセットタイプ、入力値検証設定ファイルパス、ルールセット名を指定する。

表 3 ビジネスロジックのクラス属性一覧

クラス属性	パラメータ名	必須	内容
ControllerInfo	RequestTypeName	-	ビジネスロジックを実行するリクエストコントローラを特定するリクエストタイプ名。デフォルト値は、”Normal”となる。 *リクエストタイプ名の種類は本機能説明書の内部構成を参照すること。
	InputDataSetType	○	ビジネスロジックの入力データセットの型。
ValidationInfo	ValidationFilePath	-	入力値検証設定ファイルパス。
	RuleSet	-	ルールセット名。

```
[ControllerInfo(RequestType=RequestTypeNames.NORMAL,
                InputDataSetType=typeof(SampleDs))]
[ValidationInfo(ValidationFilePath=@"Config¥validation.config",
                RuleSet="Default")]
public class SampleA : IBLogic
{
    public BLogicResult Execute(BLogicParam param)
    {
        SampleDs sample = param.ParamData as SampleDs;
        return new BLogicResult(BLogicResult.SUCCESS);
    }
}
```

リスト 4 IBLogic インタフェース実装例

■ 内部構成

◆ リクエストコントローラの生成

(1) リクエスト名の取得

クライアントからのリクエストの HTTP ヘッダのキー RequestName の値をビジネスロジック名として取得する。

(2) ビジネスロジッククラスのクラス属性の取得

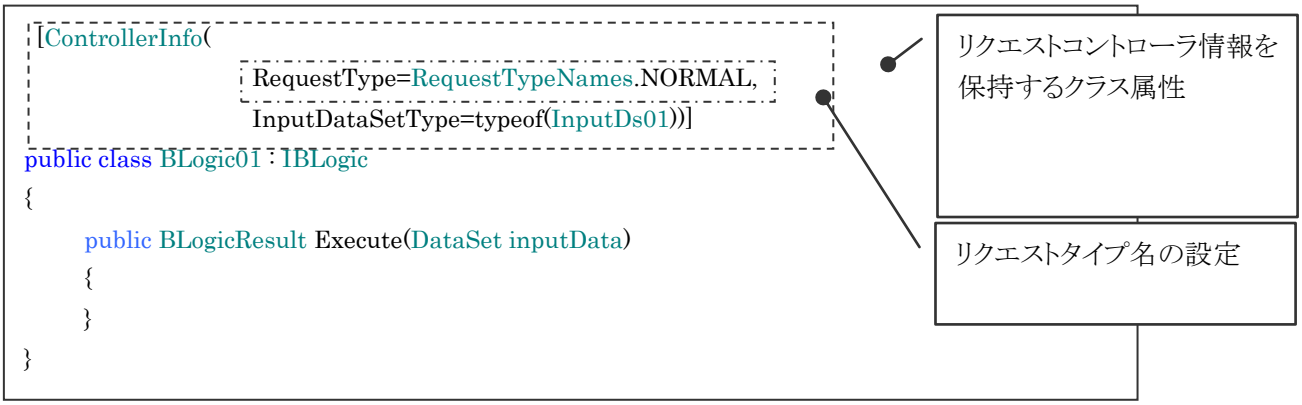
ビジネスロジック生成機能を利用して、ビジネスロジック名からビジネスロジック実装クラスのクラス属性を取得する。

以下の場合、リクエスト不正例外(InvalidRequestException)が発生する。

- HTTP ヘッダに RequestName キーが存在しない場合
- RequestName キーに対応するビジネスロジック名が存在しない場合

(3) リクエストタイプ名の取得

取得したクラス属性からコントローラ情報(ControllerInfo 属性)に設定されたリクエストタイプ名(RequestType)を取得する。



リスト 5 リクエストタイプ名の定義例

(4) リクエストコントローラの生成

取得したリクエストタイプ名に応じたリクエストコントローラを生成する。リクエストコントローラの生成に失敗した場合、TERASOLUNA.Fw.Web.Controller.UnknownRequestController を生成する。リクエストタイプ名とリクエストコントローラの既定での対応を下表に示す。

表 4 リクエストタイプ名とリクエストコントローラの対応

リクエストタイプ名	リクエストコントローラクラス
なし	TERASOLUNA.Fw.Web.Controller.BLogicRequestController, TERASOLUNA.Fw.Web
Normal	
Download	TERASOLUNA.Fw.Web.Controller.FileDownloadRequestController, TERASOLUNA.Fw.Web
Upload	TERASOLUNA.Fw.Web.Controller.FileUploadRequestController, TERASOLUNA.Fw.Web
Multipart Upload	TERASOLUNA.Fw.Web.Controller.MultipartUploadRequestController, TERASOLUNA.Fw.Web
Unknown	TERASOLUNA.Fw.Web.Controller.UnknownRequestController, TERASOLUNA.Fw.Web

◆ リクエストコントローラの実行

リクエストコントローラは、HTTP ヘッダの検証、HTTP ボディの解析、入力値検証、ビジネスロジックの生成と実行、レスポンスの設定を行う。

(1) HTTP ヘッダの検証

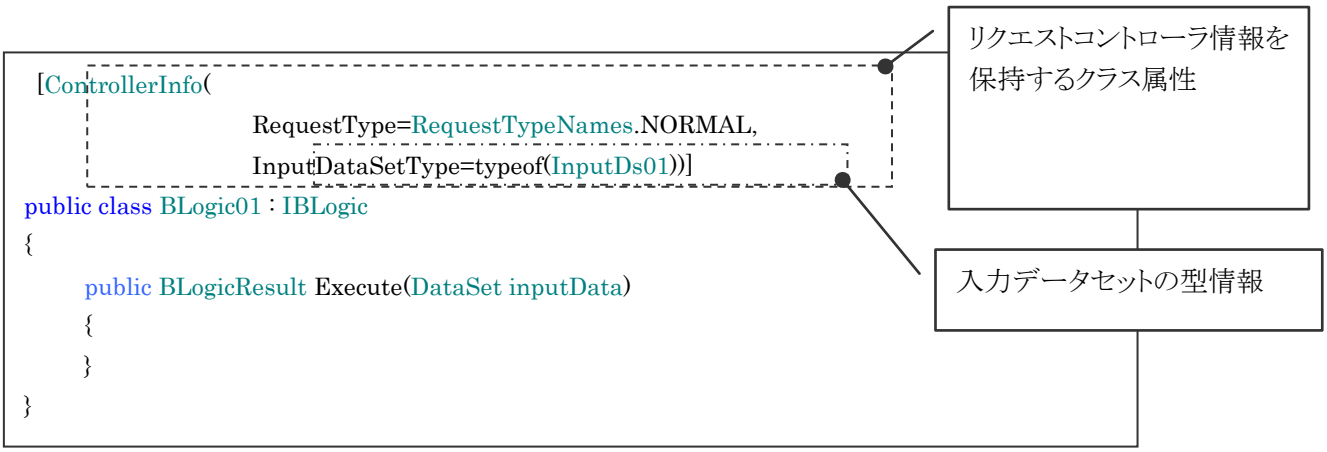
クライアントから送信された Content-Type ヘッダが ”text/xml;charset=utf-8” と設定されているか検証する。

以下の場合、リクエスト不正例外(InvalidRequestException)が発生する。

- Content-Type ヘッダがない場合
- Content-Type ヘッダに空文字が設定されている場合
- Content-Type ヘッダのフォーマットが不正の場合
- MediaType が “text/xml” でない場合
- MediaType が “text/xml”であるが、Charset が “utf-8” でない場合

(2) HTTP ボディの解析

ビジネスロジッククラスのクラス属性(ControllerInfo)の InputDataSetType に設定された入力データセットの型情報と HTTP ボディの XML データにより、ビジネスロジックに入力値として渡す型付きデータセットを生成する。



リスト 6 入力データセット型の定義例

以下の場合、フレームワーク例外(TerasolunaException)をスローする。

- ビジネスロジックのクラス属性に入力データセットの型情報が設定されていない場合
- 型付きデータセットを生成できなかった場合

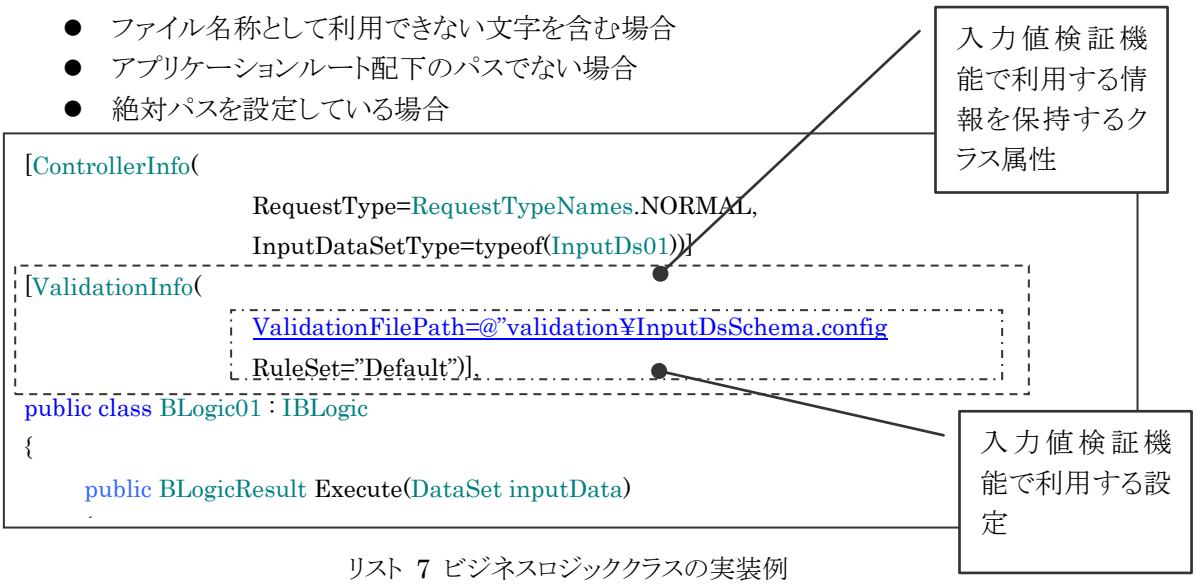
以下の場合、リクエスト不正例外(InvalidRequestException)をスローする。

- HTTP ボディに格納されているデータが XML 形式でない場合
- HTTP ボディに格納されているデータが生成対象データセットのスキーマと合致しない場合

(3) 入力値検証

ビジネスロジッククラスのクラス属性(ValidationInfo)の ValidationFilePath に設定された入力値検証設定ファイルのファイルパスを取得する。RuleSet に設定されたルールセット名を取得する。取得した入力値検証設定ファイルパスとルールセット名を利用して、入力値検証機能を実行する。ビジネスロジッククラスのクラス属性に設定されたファイルパスが以下の場合、フレームワーク例外(TerasolunaException)をスローする。

- 入力値検証設定ファイルが存在しない場合
- ファイル名称としての上限文字数を超過する場合
- ファイル名称として利用できない文字を含む場合
- アプリケーションルート配下のパスでない場合
- 絶対パスを設定している場合



リスト 7 ビジネスロジッククラスの実装例

(4) ビジネスロジックの生成と実行

入力値検証でエラーがない場合、ビジネスロジックを生成して実行する。以下の場合、TerasolunaException をスローする。

- ビジネスロジックをインスタンス化できない場合
- ビジネスロジックが IBLogic インタフェースを実装していない場合
- ビジネスロジックの返却値であるビジネスロジック結果オブジェクトが null である場合

(5) レスポンスの設定

クライアントへ返却するレスポンスを書き込む。HTTP ヘッダ、HTTP ボディに格納される情報は、以下の 4 パターンで異なる。

- 実行したビジネスロジックが正常終了した場合
- ビジネスロジックにおいて業務エラーが発生した場合
- システムエラーが発生した場合
- 入力値検証エラーが発生した場合

以下、各パターンにおける HTTP ヘッダ、ならびに HTTP ボディの一覧とサンプルのレスポンス XML を示す。

表 5 サーバ処理終了パターンと返却されるレスポンスの対応表

パターンID	終了パターン	HTTP ヘッダ		HTTP ボディ			
		キー	値	XPath			説明
①	ビジネスロジック 正常終了	content-type	text/xml; charset=utf-8	(省略)			ビジネスロジック結果オブジェクトに設定された DataSet を XML 化したもの
		status-code	200				
		status-description	OK				
②	業務エラー発生時[1]	content-type	text/xml; charset=utf-8	<errors>			ルートノード
		status-code	200		<error>		エラー情報 ※複数可
		exception	ビジネスロジック結果オブジェクトに設定された結果文字列(ResultString)			<error-message>	ビジネスロジック結果オブジェクトに設定されたエラーメッセージ
						<error-code>	ビジネスロジック結果オブジェクトに設定されたエラーメッセージのキー

[1] 業務エラーは、ResultString と Errors を設定したビジネスロジック結果オブジェクトを返却してビジネスロジックを終了した場合を指す。

パターンID	終了パターン	HTTP ヘッダ		HTTP ボディ			
		キー	値	XPath			説明
③	システムエラー発生時	content-type	text/xml; charset=utf-8	<errors>			ルートノード
		status-code	200		<error>		エラー情報 ※複数可
		status-description	OK			<error-message>	発生した例外のメッセージ
		exception	exception			<error-code>	後述するエラーコードマッピング処理で取得したエラーコード[2]
④	入力値検証エラー発生時	content-type	text/xml; charset=utf-8	<errors>			ルートノード
		status-code	200		<error>		エラー情報 ※複数可
		status-description	OK			<error-message>	入力値検証エラーメッセージ
		exception	validateException			<error-code>	入力値検証エラーエラーコード
						<error-field>	入力値検証エラー発生箇所

[2] エラーコードマッピング機能を利用しない、エラーコードを取得できない場合には、
”E_UNKNOWN_EXCEPTION” が設定される

```
<?xml version="1.0" encoding="utf-8" ?>
<OutputDS01>
  <User>
    <Name>山田太郎</Name>
    <Age>1</Age>
  </User>
</OutputDS01>
```

リスト 8 ビジネスロジック正常終了時のサンプルレスポンス電文

```
<?xml version="1.0" encoding="utf-8" ?>
<errors>
  <error>
    <error-message>既に登録された会員です。名前：山田太郎
</error-message>
    <error-code>MULTIPLE_REGISTER</error-code>
  </error>
</errors>
```

リスト 9 業務エラー時のサンプルレスポンス電文

```
<?xml version="1.0" encoding="utf-8" ?>
<errors>
  <error>
    <error-message>DBに接続できません。</error-message>
    <error-code>DB_DISCONNECTED</error-code>
  </error>
</errors>
```

リスト 10 システムエラー時のサンプルレスポンス電文

```
<?xml version="1.0" encoding="utf-8" ?>
<errors>
  <error>
    <error-message>文字列カラムの形式が正しくありません。
</error-message>
    <error-code>PATTERN</error-code>
    <error-field>User[1]/Name</error-field>
  </error>
</errors>
```

リスト 11 入力値検証エラー時のサンプルレスポンス電文

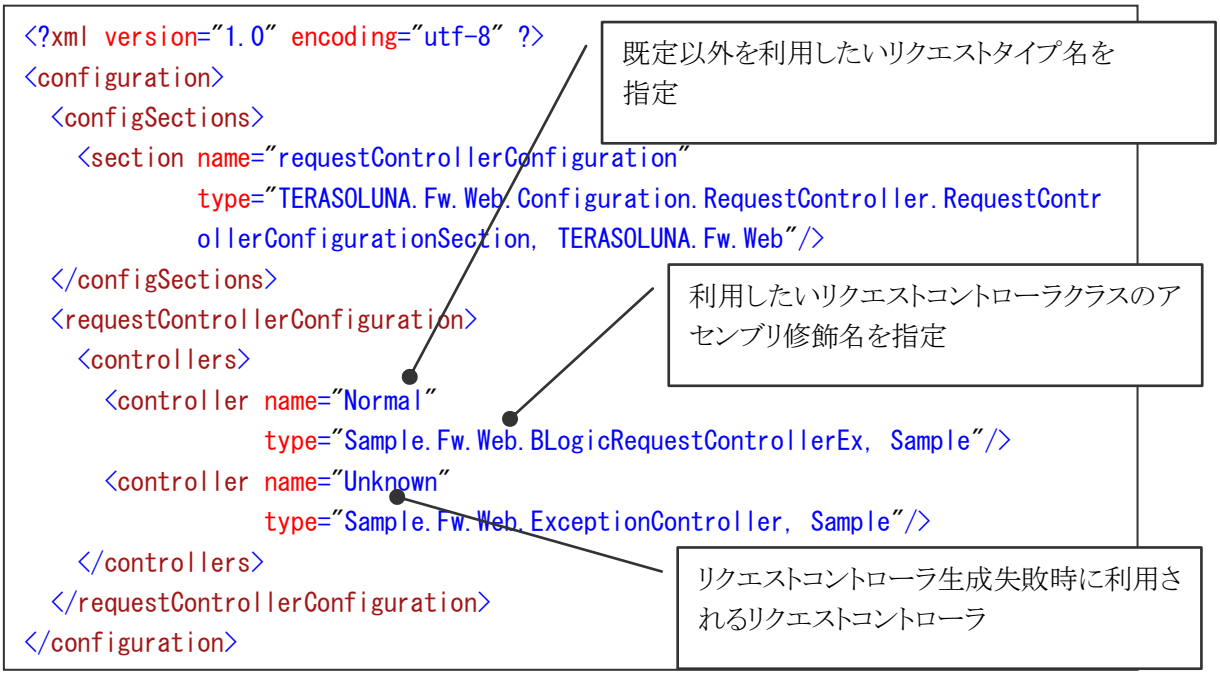
■ 拡張ポイント

◆ 既定リクエストタイプ名に対応するリクエストコントローラを変更する

既定リクエストタイプ名(Normal, Upload, Download, Unknown)のリクエストコントローラを変更する場合、Web 構成ファイル(web.config)に既定のリクエストタイプ名とリクエストコントローラのアセンブリ修飾名を定義する。

表 6 リクエストコントローラの設定

ノード	属性	必須	値
/configuration/configSections/section	name	-	構成要素名。 固定値、以下を指定する。 requestControllerConfiguration
	type	-	構成設定の処理を行う構成セクションハンドラクラス名。以下の固定値を指定。 TERASOLUNA.Fw.Web.Configuration.RequestController, TERASOLUNA.Fw.Web
/configuration/requestControllerConfiguration/controllers/controller		○	複数可
	name	○	リクエストタイプ名。
	type	○	リクエストタイプ名に紐付くリクエストコントローラのアセンブリ修飾名。




```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="requestControllerConfiguration"
             type="TERASOLUNA.Fw.Web.Configuration.RequestController.RequestContr
             ollerConfigurationSection, TERASOLUNA.Fw.Web"/>
  </configSections>
  <requestControllerConfiguration>
    <controllers>
      <controller name="Normal"
                  type="Sample.Fw.Web.BLogicRequestControllerEx, Sample"/>
      <controller name="Unknown"
                  type="Sample.Fw.Web.ExceptionController, Sample"/>
    </controllers>
  </requestControllerConfiguration>
</configuration>
```

リスト 12 Web 構成ファイル(リクエストタイプ名の設定例)

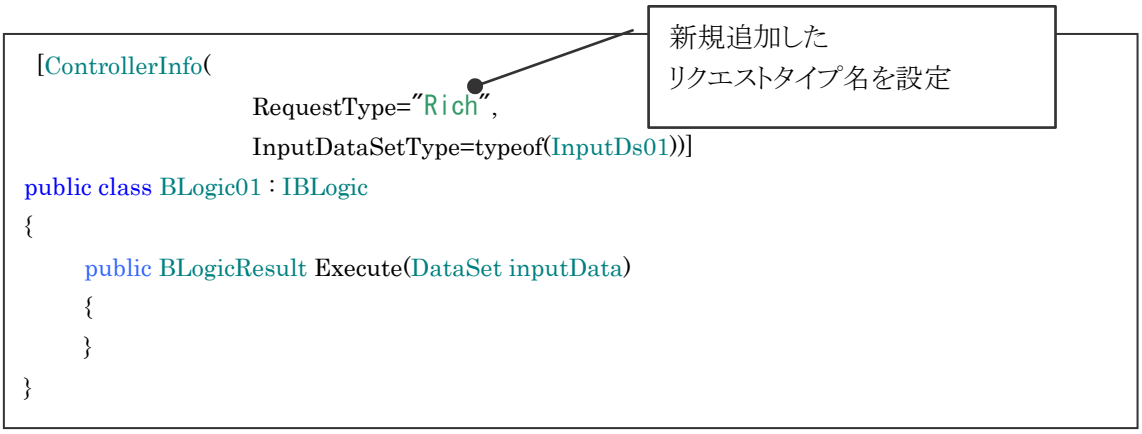
◆ 新規リクエストタイプ名に対応するリクエストコントローラを追加する

前述の「既定リクエストタイプ名のリクエストコントローラを変更する場合」と同様に、Web 構成ファイルに新規追加を行いたいリクエストタイプ名とリクエストコントローラのアセンブリ修飾名を設定する。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <requestControllerConfiguration>
    <controllers>
      <controller name="Rich"
                  type="Sample.Fw.Web.RichController, Sample"/>
    </controllers>
  </requestControllerConfiguration>
</configuration>
```

リスト 13 Web 構成ファイルの設定例(独自リクエストタイプ名)

新規に追加したリクエストタイプ名を利用するビジネスロジックの実装例を以下に示す。




```
[ControllerInfo(
                    RequestType="Rich",
                    InputDataSetType=typeof(InputDs01))]
public class BLogic01：IBLogic
{
    public BLogicResult Execute(DataSet inputData)
    {
    }
}
```

リスト 14 ビジネスロジック実装例(独自リクエストタイプ名)

◆ エラー時にクライアントに返却する電文フォーマットを変更する

(1) ルートノードを変更する場合

ルートノード(既定では、”error”)を変更する場合、フレームワークが提供するBLogicRequestController の CreateInitializeErrorDocument メソッドをオーバライドし、任意に実装を行う。

(2) ルートノード配下のノード名を変更する場合

ルートノード配下のノード名(既定では、”error”, “error-message”, “error-code”)を変更する場合、BLogicRequestController の以下のメソッドをオーバライドし、任意に実装を行う。

表 7 エラー電文を作成するメソッド一覧

メソッド	呼び出される契機
WriteBLogicErrorResponseBody	業務エラー
WriteValidationErrorResponseBody	入力値検証エラー時
WriteSystemErrorResponseBody	システムエラー時

◆ クライアントに返却する HTTP ヘッダに任意の値を追加する

BLogicRequestController の以下のメソッドをオーバライドし、任意に実装を行う。

表 8 ヘッダを作成するメソッド一覧

メソッド	呼び出される契機
WriteSuccessXmlResponseHeader	業務正常終了時
WriteBLogicErrorResponseHeader	業務エラー時
WriteValidationErrorResponseHeader	入力値検証エラー時
WriteSystemErrorResponseHeader	システムエラー時

■ 関連機能

- CM-02 入力値検証機能
- CM-04 ビジネスロジック生成機能
- WB-02 ファイルアップロード機能
- WB-03 ファイルダウンロード機能

WB-02 ファイルアップロード機能

■ 概要

本機能は、クライアントからのファイルアップロード要求に対応するファイルの受信処理及びビジネスロジックを実行し、クライアントへ XML でファイルアップロード結果を通知する機能である。クライアントからのファイルアップロード要求に対応できる形式は、マルチパート形式とバイナリ形式の 2 種類である。

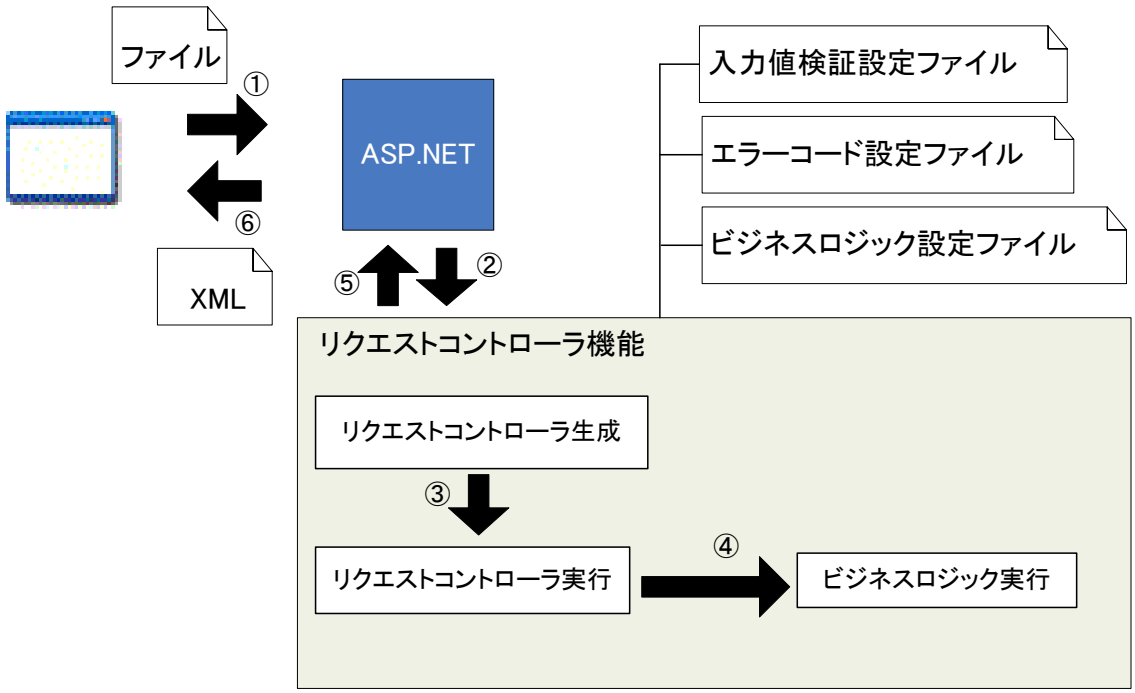


図 1 ファイルアップロード機能

クライアントから送信するファイルアップロード要求に基づいて、実行するリクエストコントローラを生成する。生成したリクエストコントローラがファイルの受信処理およびビジネスロジックを実行する。リクエストコントローラの処理の途中で例外が発生した場合、例外に対応したエラーコードをエラーコード設定ファイルから取得し、エラー電文(XML)にエラーコードを含めてクライアントへ送信する。

リクエストコントローラに関連する機能として、ビジネスロジックは TERASOLUNA で提供しているビジネスロジック生成機能を利用する。ビジネスロジック機能の詳細な内容は、「CM-4 ビジネスロジック生成機能」を参照すること。本機能で説明するリクエストコントローラ機能は、クライアントからのファイルアップロード要求に対してファイルを受信するリクエストコントローラである。クライアントとサーバで XML を送受信する処理に対応するリクエストコントローラは、「WB-01 リクエストコントローラ機能」、クライアントからのファイルダウンロード要求に対して、ファイルを送信するリクエストコントローラは、「WB-03 ファイルダウンロード機能」を参照すること。

■ 使用方法(マルチパート形式)

Web 構成ファイル、ビジネスロジック設定ファイル、エラーコード設定ファイルは、リクエストコントローラ機能と同様の設定を行う。これらの設定については、「WB-01 リクエストコントローラ機能」を参照のこと。ここでは、ファイルアップロード機能のマルチパート形式を利用する場合に必要なビジネスロジックの実装方法について説明する。

◆ 実装方法

ビジネスロジックは必ず IUploadBLogic インタフェースを実装する。IUploadBLogic 実装クラスのクラス属性に必ずリクエストタイプ名”MultipartUplaod”を指定する(表 1)。クライアントからマルチパート形式で送信したデータは、IUploadBLogic で実装する MultipartItemList プロパティで取得することができる。

表 1 ビジネスロジックのクラス属性一覧(マルチパート形式)

クラス属性	パラメータ名	必須	内容
ControllerInfo	RequestTypeName	○	ビジネスロジックを実行するリクエストコントローラを特定するリクエストタイプ名。 以下の固定地を指定。 MultipartUpload

IUPloadBLogic を実装する。

```
[ControllerInfo(RequestType = RequestTypeNames.MULTIPART_UPLOAD)]
public class BLogic01 : IUploadBLogic
{
    private IDictionary<string, IMultipartItem> _multipartItemList =
                                    new Dictionary<string, IMultipartItem>();

    public IDictionary<string, IMultipartItem> MultipartItemList
    {
        get
        {
            return _multipartItemList;
        }
        set
        {
            _multipartItemList = value;
        }
    }

    public BLogicResult Execute(BLogicParam param)
    {
        foreach (IMultipartItem item in _multipartItemList.Values)
        {
            if (!item.IsText)
            {
                MultipartFileItem fileItem = item as MultipartFileItem;
                using (FileStream stream = new FileStream(
                        @"C:¥temp¥" + fileItem.Filename, FileMode.Create))
                {
                    byte[] buffer = new byte[256];
                    int i = 0;
                    while (fileItem.OutputStream.Read(buffer, 0, 256) != 0)
                    {
                         stream.Write(buffer,0, 256);
                    }
                    stream.Flush();
                }
            }
        }
        return new BLogicResult(BLogicResult.SUCCESS, new DataSet());
    }
}
```

呼び出し元のリクエストコントローラに公開する設定情報 RequestType には RequestTypeName.MULTIPART_UPLOAD を指定する。

マルチパートアップローデータを設定・取得するプロパティを実装する。

マルチパートアップロードリクエストコントローラが、リクエストを受け付けるたびに新しいインスタンスを設定する。

マルチパートデータがテキストデータかどうかチェック。

リスト 1 ビジネスロジックの実装例(マルチパート形式)

■ 使用方法(バイナリ形式)

ビジネスロジック設定ファイル、エラーコード設定ファイルは、リクエストコントローラ機能と同様の設定を行う。これらの設定については、「WB-01 リクエストコントローラ機能」を参照のこと。ここでは、ファイルアップロード機能のバイナリ形式を利用する場合に必要な Web 構成ファイル、ビジネスロジックの実装方法について説明する。

◆ Web 構成ファイル

本機能のバイナリ形式アップロードを利用する場合、Web 構成ファイル(web.config)にバイナリアップロード時の一時保存ルートディレクトリの指定及びアップロードファイルの最大サイズを定義することができる。一時保存ルートディレクトリを定義しない場合、システムの一時フォルダを利用する。また、アップロードファイルの最大サイズを定義しない場合、アップロードファイルの最大サイズは 4MB となる。

表 2 Web 構成ファイル

ノード	属性	必須	値
/configuration/appSettings/add	key	-	キー名。 固定値、以下を指定する。 “RootTmpDirectory”
	value	-	一時保存ルートディレクトリのディレクトリパスを絶対パスで指定する。省略した場合は、現在のシステムの一時フォルダのパスを利用する。
/configuration/system.web/httpRuntime	maxRequestLength	-	アップロードファイルの最大サイズ(KByte)を数値型で指定する。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  …
  <appSettings>
    <!-- バイナリアップロード時の一時フォルダの指定  -->
    <add key="RootTmpDirectory" value="C:¥MyFolder¥CustomTemp"/>
  </appSettings>
  …
  <system.web>
    <httpRuntime maxRequestLength="32768"/>
  </system.web>
  …
```

リスト 2 Web 構成ファイルの記述例

◆ 実装方法

ビジネスロジックは必ず IBLogic インタフェースを実装する。IBLogic 実装クラスのクラス属性に必ずリクエストタイプ名”Upload”を指定する(表 3)。IBLogic 実装クラスの入力データセット(BLogicParam の ParamData プロパティ)は必ず TERASOLUNA で提供しているファイルアップロード用データセット(UploadFileInfo クラス)となる(表 4)。バイナリ形式でアップロードしたデータのファイルパスは、ファイルアップロード用データセット(UploadFileInfo クラス)の TempFiles テーブルの AbsolutePath データカラムから取得することができる(表 4)。

表 3 ビジネスロジックのクラス属性一覧(バイナリ形式)

クラス属性	パラメータ名	必須	内容
ControllerInfo	RequestTypeName	○	ビジネスロジックを実行するリクエストコントローラ情報。 以下の固定地を指定。 Upload を指定。

表 4 ファイルアップロード用データセットのテーブル一覧

データテーブル名	データカラム名	データ型	説明
ApplicationRoot	AbsolutePath	System.String	アプリケーションルートの絶対パス。
TempFiles	AbsolutePath	System.String	一時保存ディレクトリに保存したアップロードファイルの絶対パス。

```
[ControllerInfo(RequestType = RequestTypeNames.UPLOAD)]
public class BLogic01 : IBLogic
{
    public BLogicResult Execute(BLogicParam blogicParam)
    {
        // 入力データセットのキャスト
        UploadFileInfo inputDs = (UploadFileInfo)blogicParam.ParamD
        // アプリケーションルートの絶対パスを取得
        string appRootPath = inputDs.ApplicationRoot[0].AbsolutePath;
        // 一時保存されているアップロードファイルのパスを取得
        string tempFilePath = inputDs.TempFiles[0].AbsolutePath;
        // 一時保存されているアップロードファイルのファイル情報を作成
        FileInfo tempFileInfo = new FileInfo(tempFilePath);

        // 一時保存されているアップロードファイルをアップロードファイル用
        // ディレクトリに移動
        File.Move(tempFilePath, appRootPath + @"¥Upload¥"
                                            + tempFileInfo.Name);
        // ビジネスロジック結果オブジェクトの生成
        return new BLogicResult(BLogicResult.SUCCESS, new DataSet());
    }
}
```

呼び出し元のリクエストコントローラに公開する設定情報 RequestType には RequestTypeName.UPLOAD を指定する。

入力データセットは、UploadFileInfo となる。

リスト 3 ビジネスロジックの実装例(バイナリ形式)

■ 内部構成(マルチパート形式)

◆ リクエストコントローラの生成

マルチパートアップロードコントローラの生成は、リクエストコントローラ機能と同様の処理が行われる。これらの処理の詳細については、「WB-01 リクエストコントローラ機能」を参照のこと。

◆ リクエストコントローラの実行

マルチパートアップロードコントローラは、HTTP ヘッダの解析、HTTP ボディの解析、ビジネスロジックの生成と実行、レスポンスの生成を行う。

(1) HTTP ヘッダの解析

Content-Type ヘッダから boundary 属性の値と、charset 属性の値を取得する (表 5)。

表 5 HTTP ヘッダから取得する情報一覧

取得する情報	内容
boundary	マルチパートアップロードデータのパートごとの区切りを表す文字列。
charset	マルチパートアップロードデータをエンコードした際に使用する文字コード。

以下の場合、マルチパートアップロードコントローラは InvalidRequestException をスローする。

- boundary 属性がない場合
- charset 属性の値が不正な場合

(2) HTTP ボディの解析

複数のアップロードデータが存在する場合、それぞれのアップロードデータに対して以下の処理を行う。

各パートのヘッダから、Content-Disposition、Content-Type を取得する(表 6)。

表 6 取得するヘッダ情報一覧

取得するヘッダ情報		内容
Content-Disposition	name	マルチパート要素名。
	filename(アップロードデータがファイルである場合)	クライアント側で送信するファイルの名前。
Content-Type		パートごとのアップロードデータの形式。

以下の場合、マルチパートアップロードコントローラは InvalidRequestException をスローする。

- マルチパート要素名が重複している場合
- Content-Disposition ヘッダがない場合
- Content-Type ヘッダがない場合
- アップロードデータがファイルで、filename 属性がない場合

各パートの Content-Type ヘッダが、”application/x-www-form-urlencoded”の場合はアップロードデータがテキストであるため、MultipartTextItem クラス(表 8)のインスタンスを生成する。”application/octet-stream”の場合はアップロードデータがファイルであるため、MultipartFileItem クラス(表 9)のインスタンスを生成し、アップロードデータの一時ファイルを生成する。一時ファイルは、ガーベジコレクションが MultipartFileItem インスタンスを破棄するタイミングで削除する。

表 7 Content-Type よって作成される MultipartItem クラス一覧

Content-Type の値	格納する MultipartItem クラス
application/x-www-form-urlencoded	MultipartTextItem クラス。
application/octet-stream	MultipartFileItem クラス。

(3) ビジネスロジックの生成と実行

呼び出し対象ビジネスロジックの型を取得し、インスタンス化する。

以下の場合、マルチパートアップロードコントローラは TerasolunaException をスローする。

- 呼び出し対象ビジネスロジックをインスタンス化できない場合
- 呼び出し対象ビジネスロジックが IUploadBLogic インタフェースを実装していない場合

生成した IUpload 実装インスタンスの MultipartItemList プロパティに生成した MultipartItem のリストを設定し、IUpload 実装インスタンスを実行する。

(4) レスポンスの設定

マルチパートアップロードコントローラが実行するレスポンスの設定は、リクエストコントローラ機能と同様の処理が行われる。これらの処理の詳細については、「WB-01 リクエストコントローラ機能」を参照のこと。

■ 内部構成(バイナリ形式)

◆ リクエストコントローラの生成

ファイルアップロードコントローラの生成は、リクエストコントローラ機能と同様の処理が行われる。これらの処理の詳細については、「WB-01 リクエストコントローラ機能」を参照のこと。

◆ リクエストコントローラの実行

ファイルアップロードコントローラは、HTTP ヘッダの検証、HTTP ボディの解析、ビジネスロジックの生成と実行、レスポンスの生成を行う。

(1) HTTP ヘッダの検証

Content-Type ヘッダに”application/octet-stream”が適切に設定しているか検証する。

以下の場合、ファイルアップロードコントローラは InvalidRequestException をスローする。

- Content-Type ヘッダがない場合
- Content-Type ヘッダに空文字列を設定している場合
- Content-Type ヘッダのフォーマットが不正な場合
  - MediaType が”application/octet-stream”でない場合

Content-Disposition ヘッダに入っている文字列をデコードし、ファイル名称として取得する。取得したファイル名称を HTTP コンテキストに設定する。

以下の場合、ファイルアップロードコントローラは InvalidRequestException をスローする。

- Content-Disposition ヘッダがない場合
- Content-Disposition ヘッダに空文字列を設定している場合

- Content-Disposition ヘッダのフォーマットが不正の場合
  - Base64 でエンコードしていない場合
  - Base64 でエンコードしたファイル名称の長さが 4 の倍数でない場合
- Content-Disposition ヘッダの値のフォーマットが不正等の理由でファイル名称を取得できなかった場合

(2) HTTP ボディの解析

① アップロードファイルの一時保存ルートディレクトリの生成

Web 構成ファイル(web.config)に一時保存ルートディレクトリのパスを指定する。Web 構成ファイルに一時保存ルートディレクトリのパスを指定しない場合は、システムの一時フォルダを利用する。

Web 構成ファイルの /configuration/appSettings/add 要素の key 属性に RootTmpDirectory、value 属性に一時保存ルートディレクトリの絶対パスを設定する。

```
<configuration>
 <appSettings>
  <add key="RootTmpDirectory" value="C:¥MyFolder¥CustomTemp"/>
 </appSettings>
</configuration>
```

リスト 4 一時保存ルートディレクトリの設定例

Web 構成ファイルに設定している一時保存ルートディレクトリパスが以下の場合、ファイルアップロードコントローラが ConfigurationErrorsException をスローする。

- 空文字列を設定している場合
- ディレクトリパスとして設定できるサイズを超過する場合
- ディレクトリ名称として指定できない文字を含む場合

② アップロードファイルの一時保存ディレクトリの生成

アップロードファイルの一時保存ルートディレクトリ直下に乱数で生成した文字列をディレクトリ名としてディレクトリを生成する。

以下の場合、ファイルアップロードコントローラは TerasolunaException をスローする。

- 一時保存ルートディレクトリにディレクトリ作成権限がない場合

③ アップロードファイルの一時保存

アップロードファイルの一時保存ディレクトリにクライアントからバイナリ形式で送信してきたアップロードファイルを保存する。

以下の場合、ファイルアップロードコントローラは TerasolunaException をスローする。

- 一時保存ディレクトリにファイル作成権限がない場合

図 2 Web 構成ファイルに一時保存ルートディレクトリを利用する場合のフォルダ構成

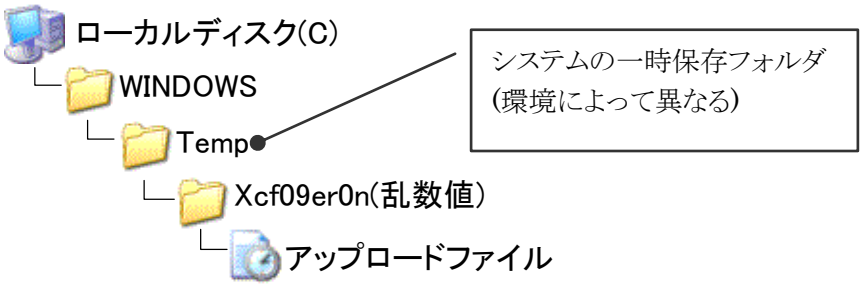


図 3 既定の一時保存ルートディレクトリを利用する場合のフォルダ構成

④ ファイルアップロード用データセットの生成

ビジネスロジックの入力データセットは、ファイルアップロード用データセット(UploadFileInfo インスタンス)を生成し、HTTP コンテキストに設定する。

ファイルアップロード用データセット(UploadFileInfo インスタンス)は DataSet を継承した型付データセットで表 4 のデータテーブルとデータカラムを持つ。

(3) ビジネスロジックの生成と実行

ビジネスロジックを実行する。ビジネスロジックの入力データセットは、HTTP コンテキストで設定したファイルアップロード用データセット(UploadFileInfo インスタンス)である。

以下の場合、リクエストコントローラは TerasolunaException をスローする。

- 呼び出し対象ビジネスロジックをインスタンス化できない場合
- 呼び出し対象ビジネスロジックが IBLogic インタフェースを実装していない場合
- ビジネスロジックの返却値であるビジネスロジック結果オブジェクトが null である場合

(4) レスポンスの設定

レスポンスを書き込む前に、一時保存ディレクトリを削除する。

以下の場合、リクエストコントローラは TerasolunaException をスローする。

- 一時保存ルートディレクトリに削除権限がない場合
- 一時保存ルートディレクトリ配下に、現在のプロセス、および他のプロセスで書き込みを禁止している、もしくは開いているファイルが存在する場合

正常に一時保存ディレクトリを削除したのち、リクエストコントローラ機能のレスポンスの設定と同様の処理を実行する。詳細については、「WB-01 リクエストコントローラ機能」を参照のこと。

■ 拡張ポイント(バイナリ形式)

◆ アップロードファイル名のデコード方式を変更する

アップロードファイル名のデコード方式(既定では、Base64 エンコーディングによるエンコードによるファイル名のデコーディングを想定)を変更する場合、FileUploadRequestController クラスの DecodeFileName メソッドをオーバーライドし、任意に実装を行う。

■ 関連機能

- CM-04 ビジネスロジック生成機能
- WB-01 リクエストコントローラ機能

WB-03 ファイルダウンロード機能

■ 概要

本機能は、クライアントのリクエスト要求に対応する入力値検証、ビジネスロジックを実行し、クライアントへファイルを送信する機能である。

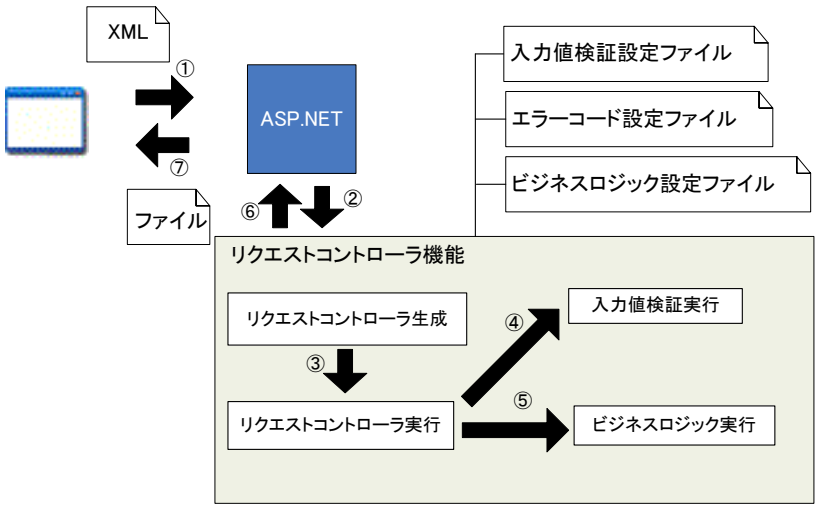


図 1 ファイルダウンロード機能

クライアントから送信されるリクエスト情報に基づいて、実行するリクエストコントローラを生成する。生成されたリクエストコントローラが入力値検証、ビジネスロジックを実行する。ビジネスロジック実行後、リクエストコントローラはビジネスロジックの結果に基づいて、クライアントへファイルをバイナリ形式で送信する。リクエストコントローラの処理の途中で例外が発生した場合は、例外に対応したエラーコードをエラーコード設定ファイルから取得し、エラー電文(XML)にエラーコードを含めてクライアントへ送信する。

リクエストコントローラに関連する機能として、入力値検証は、TERASOLUNA で提供している入力値検証機能、ビジネスロジックは TERASOLUNA で提供しているビジネスロジック生成機能を利用する。入力値検証機能、ビジネスロジック機能の詳細な内容は、「CM-02 入力値検証機能」、「CM-04 ビジネスロジック生成機能」を参照すること。本機能で説明するリクエストコントローラ機能は、クライアントからのファイルダウンロード要求に対して、ファイルを送信するリクエストコントローラである。クライアントとサーバで XML を送受信する処理に対応するリクエストコントローラは、「WB-01 リクエストコントローラ機能」、クライアントからのファイルアップロード要求に対して、ファイルを受信するリクエストコントローラは、「WB-02 ファイルアップロード機能」を参照すること。

■ 使用方法

Web 構成ファイル、ビジネスロジック設定ファイル、エラーコード設定ファイル、入力値検証設定ファイルは、リクエストコントローラ機能と同様の設定を行う。これらの設定については、「WB-01 リクエストコントローラ機能」を参照のこと。ここでは、ファイルダウンロード機能の利用に必要なビジネスロジックの実装方法について説明する。

◆ 実装方法

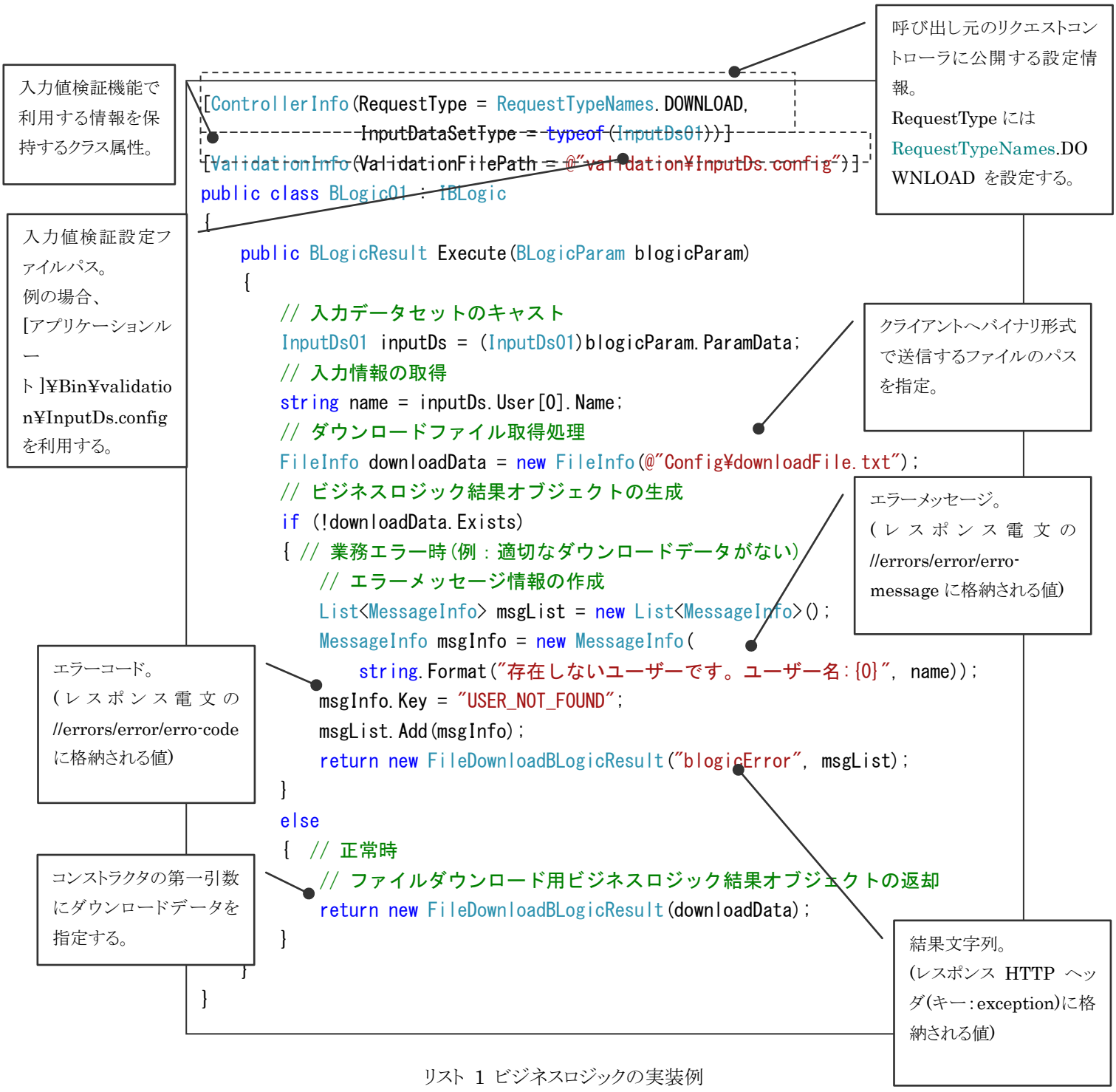
IBLogic インタフェースを実装し、実装したクラスのクラス属性にリクエストタイプ名、入力データセットタイプ、入力値検証設定ファイルパス、ルールセット名を指定する。必ずリクエストタイプ名には"Donwload"を指定する(表 1)。返却値はファイルダウンロード用ビジネスロジック結果オブジェクト(FileDownloadBLogicResult 型のインスタンス)とする。ファイルダウンロード用ビジネスロジック結果オブジェクトに設定できるダウンロードファイルデータの形式は、ファイル情報(FileInfo クラス)形式とバイト配列形式の 2 種類がある(表 2)。

表 1 ビジネスロジックのクラス属性一覧

クラス属性	パラメータ名	必須	内容
ControllerInfo	RequestTypeName	○	ビジネスロジックを実行するリクエストコントローラを特定するリクエストタイプ名。 以下の固定値を指定。 Download
	InputDataSetType	○	ビジネスロジックの入力データセットの型。
ValidationInfo	ValidationFilePath	-	入力値検証設定ファイルパス。
	RuleSet	-	ルールセット名。 デフォルト値は、Default。

表 2 FileDownloadBLogicResult のコンストラクタとその利用観点

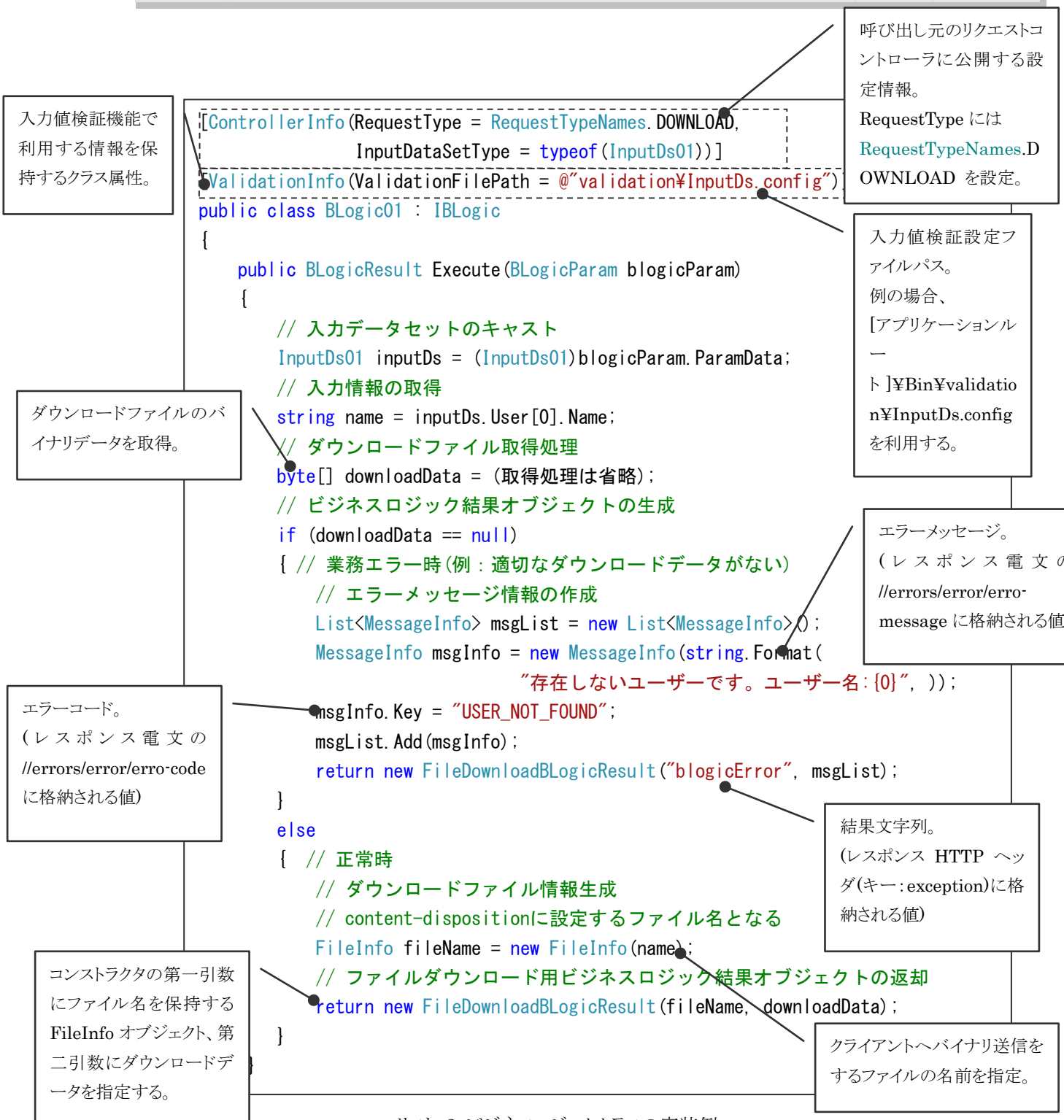
コンストラクタ	引数	引数の説明	利用観点
FileDownloadBLogicResult (FileInfo fileInfo)	fileInfo	ダウンロードファイルのパス情報を保持する FileInfo オブジェクト	Web サーバ上にあるファイルをそのままダウンロードする場合
FileDownloadBLogicResult (FileInfo fileInfo, byte[] data)	fileInfo	ダウンロードデータと対応付けるのファイル名を保持する FileInfo オブジェクト	DB やローカルにあるファイルをバイナリ形式で保持しており、そのバイナリ形式のままダウンロードする場合
	data	ダウンロードデータ	




```
[ControllerInfo(RequestType = RequestTypeNames.DOWNLOAD,
                InputDataSetType = typeof(InputDs01))]
[ValidationInfo(ValidationFilePath = @"validation¥InputDs.config")]
public class BLogic01 : IBLogic
{
    public BLogicResult Execute(BLogicParam blogicParam)
    {
        // 入力データセットのキャスト
        InputDs01 inputDs = (InputDs01)blogicParam.ParamData;
        // 入力情報の取得
        string name = inputDs.User[0].Name;
        // ダウンロードファイル取得処理
        FileInfo downloadData = new FileInfo(@"Config¥downloadFile.txt");
        // ビジネスロジック結果オブジェクトの生成
        if (!downloadData.Exists)
        { // 業務エラー時(例：適切なダウンロードデータがない)
            // エラーメッセージ情報の作成
            List<MessageInfo> msgList = new List<MessageInfo>();
            MessageInfo msgInfo = new MessageInfo(
                string.Format("存在しないユーザーです。ユーザー名:{0}", name));
            msgInfo.Key = "USER_NOT_FOUND";
            msgList.Add(msgInfo);
            return new FileDownloadBLogicResult("blogicError", msgList);
        }
        else
        {  // 正常時
            // ファイルダウンロード用ビジネスロジック結果オブジェクトの返却
            return new FileDownloadBLogicResult(downloadData);
        }
    }
}
```

リスト 1 ビジネスロジックの実装例

(ダウンロードファイル情報が FileInfo 形式の場合)




```
[ControllerInfo(RequestType = RequestTypeNames.DOWNLOAD,
                InputDataSetType = typeof(InputDs01))]
[ValidationInfo(ValidationFilePath = @"validation¥InputDs.config")]
public class BLogic01 : IBLogic
{
    public BLogicResult Execute(BLogicParam blogicParam)
    {
        // 入力データセットのキャスト
        InputDs01 inputDs = (InputDs01)blogicParam.ParamData;
        // 入力情報の取得
        string name = inputDs.User[0].Name;
        // ダウンロードファイル取得処理
        byte[] downloadData = (取得処理は省略);
        // ビジネスロジック結果オブジェクトの生成
        if (downloadData == null)
        { // 業務エラー時(例：適切なダウンロードデータがない)
            // エラーメッセージ情報の作成
            List<MessageInfo> msgList = new List<MessageInfo>();
            MessageInfo msgInfo = new MessageInfo(string.Format(
                                    "存在しないユーザーです。ユーザー名:{0}", ));
            msgInfo.Key = "USER_NOT_FOUND";
            msgList.Add(msgInfo);
            return new FileDownloadBLogicResult("blogicError", msgList);
        }
        else
        {  // 正常時
            // ダウンロードファイル情報生成
            // content-dispositionに設定するファイル名となる
            FileInfo fileName = new FileInfo(name);
            // ファイルダウンロード用ビジネスロジック結果オブジェクトの返却
            return new FileDownloadBLogicResult(fileName, downloadData);
        }
    }
```

リスト 2 ビジネスロジッククラスの実装例

(ダウンロードファイル情報がバイト配列の形式の場合)

■ 内部構成

◆ リクエストコントローラの生成

ファイルダウンロードリクエストコントローラの生成は、リクエストコントローラ機能と同様の処理が行われる。これらの処理の詳細については、「WB-01 リクエストコントローラ機能」を参照のこと。

◆ リクエストコントローラの実行

ファイルダウンロードリクエストコントローラは、HTTP ヘッダの検証、HTTP ボディの解析、入力値検証、ビジネスロジックの生成と実行、レスポンスの生成を行う。ファイルダウンロードリクストコントローラが実行する HTTP ヘッダの検証、HTTP ボディの解析、入力値検証は、リクエストコントローラ機能と同様の処理が行われる。これらの処理の詳細については、「WB-01 リクエストコントローラ機能」を参照のこと。ここでは、ビジネスロジックの生成と実行、レスポンスの設定について解説する。

(4) ビジネスロジックの生成と実行

入力値検証でエラーがない場合、ビジネスロジックを生成して実行する。ビジネスロジックは、ビジネスロジックの実行結果として、ファイルダウンロード用ビジネスロジック結果オブジェクトを返却する。

以下の場合、リクエストコントローラは TerasolunaException をスローする。

- ビジネスロジックをインスタンス化できない場合
- ビジネスロジックが IBLogic インタフェースを実装していない場合
- ビジネスロジックの返却値であるファイルダウンロード用ビジネスロジック結果オブジェクトが null である場合
- ビジネスロジックの返却値であるファイルダウンロード用ビジネスロジック結果オブジェクトに、存在しないファイルの情報（FileInfo）が設定されている場合

(5) レスポンスの設定

クライアントへ返却するレスポンスを書き込む。ビジネスロジック正常終了時に設定される HTTP ヘッダの情報を表 3 に示す。Content-Disposition ヘッダには、ファイル名を Base64 でエンコードされた文字列を設定する。

表 3 設定される HTTP ヘッダの値

パターン ID	終了パターン	HTTP	
		キー	値
①	ビジネスロジック正常終了	content-type	application/octet-stream
		content-disposition	attachment; filename=?ISO-2022JP?B?{Base64 エンコード済みファイル名}?=
		status-code	200
		status-description	OK

また、HTTP ボディには、ファイルダウンロード用ビジネスロジック結果オブジェクトに設定されているダウンロードデータをレスポンスに書き込み、クライアントにバイナリファイルを送信する。

■ 拡張ポイント

◆ ダウンロードファイル名のエンコード方式を変更する

ダウンロードファイル名のエンコード方式(既定では、Base64 エンコーディング)を変更する場合、FileDownloadRequestController クラスの EncodeFileName メソッドをオーバーライドする。

■ 関連機能

- CM-02 入力値検証機能
- CM-04 ビジネスロジック生成機能
- WB-01 リクエストコントローラ機能

WC-01 セッション管理機能

■ 概要

本機能は、セッション情報をライフサイクルレベルで管理する機能である。ライフサイクルレベルとは、セッション情報をグループ化する単位のことである。ライフサイクルレベルごとにセッション情報を管理することで、セッション情報のグループごとの一括削除を実現する。

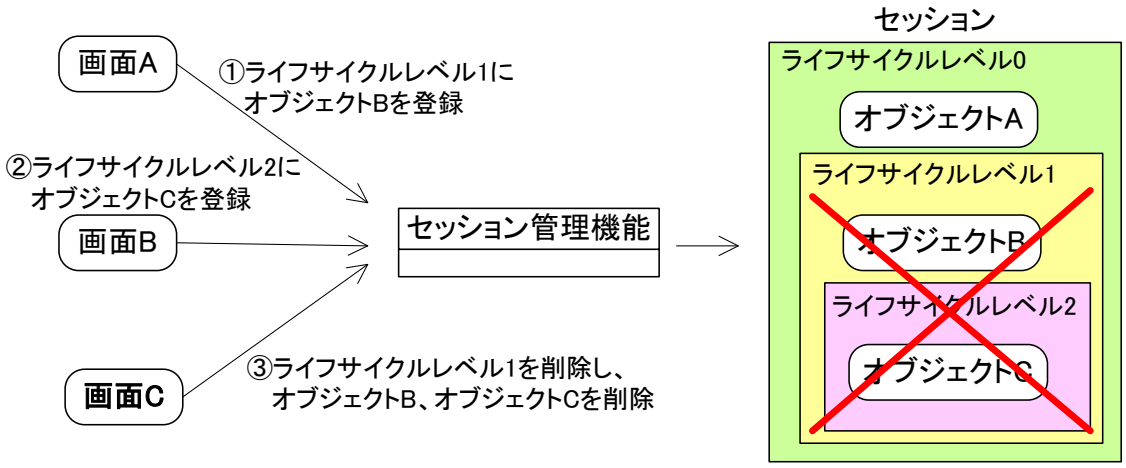


図 1 セッション管理機能

セッション情報をライフサイクルレベルごとの階層構造で管理する。ライフサイクルレベル 0 はライフサイクルレベル 1 を包含し、ライフサイクルレベル 1 はライフサイクルレベル 2 を包含する。ライフサイクルレベルは、以降同様に包含関係を形成する。

セッション内へのオブジェクトの登録は、セッションキーと登録するオブジェクトとライフサイクルレベルを指定する。

セッション内のオブジェクトの削除は、ライフサイクルレベルまたはセッションキーを指定する。ライフサイクルレベル 1 を削除すると、ライフサイクルレベル 2 で保存されているオブジェクトも削除される。このようにセッション情報をグループ単位で削除することで、セッションに登録した個別データの削除漏れを防ぐことができる。

■ 使用方法

◆ 実装方法

セッションにオブジェクトを登録する場合は、セッションキー、登録するオブジェクト、ライフサイクルレベルを指定する。

```
public partial class SC0010 : System.Web.UI.Page
{
    private SessionManager _sessionManager = null;

    public override void OnInit()
    {
        base.OnInit();
        _sessionManager = new SessionManager(this.Session);
    }

     protected void LoginButton_Click(object sender, EventArgs e)
     {
         // 業務処理　省略

         // ライフサイクルレベル0にログイン情報を登録する
        _sessionManager.Add("ID", TextBoxFirstName.Text, 0);

         // 画面遷移
        Response.Redirect("SC0020.aspx");
     }
}
```

リスト 1 セッションにオブジェクトを登録する実装例

セッション内のオブジェクトをライフサイクルレベルで削除する場合は、ライフサイクルレベルを指定する。単一オブジェクトを削除する場合は、セッションキーを指定する。

```
public partial class SC0010 : System.Web.UI.Page
{
    private SessionManager _sessionManager = null;

    public override void OnInit()
    {
        base.OnInit();
        _sessionManager = new SessionManager(this.Session);
    }

    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            // 画面初回表示時にライフサイクルレベル1の項目を削除する
            _sessionManager.Remove(1);

            // キー値”ID”で保存されているセッション情報を削除する
            _sessionManager.Remove("ID");
        }
    }
}
```

リスト 2 セッション内のオブジェクトを削除する実装例

セッション内のオブジェクトを取得する場合は、取得するセッションキーをインデックスに指定する。

```
// セッション内のオブジェクトを取得する。
SessionManager sessionManager = new SessionManager(Session);
string id = (string)sessionManager["ID"];
```

リスト 3 セッション内のオブジェクトを取得する実装例

■ 内部構成

◆ セッション情報のライフサイクルレベル管理機能

セッション管理機能では、セッション情報のライフサイクルレベル管理を「レベル管理マスタ」と「キー管理マスタ」の 2 種類のマスタデータによって管理する。レベル管理マスタはライフサイクルレベルごとのセッションキーを管理する。キー管理マスタはセッションキーごとのライフサイクルレベルを管理する。セッション管理機能では、これらのマスタデータを利用し、ライフサイクル管理やセッション情報の一括削除を実現する。

表 1 セッションに保存されるマスタデータ

セッションキー名	型	説明
SESSION_MANAGER_LEVEL_MASTER	Dictionary<int, IList<string>>	レベル管理マスタ。 ライフサイクルごとのセッションキーを管理する。Dictionary の key 値はライフサイクルレベル、value 値は管理対象のセッションキーが格納される。
SESSION_MANAGER_KEY_MASTER	Dictionary<string, int>	キー管理マスタ。 セッションキーごとのライフサイクルレベルを管理する。Dictionary の key 値は管理対象のセッションキー、value 値はライフサイクルレベルが格納される。

(1) セッション登録時の制限事項

SessionManager の Add メソッドで新規にオブジェクトを登録するときに、以下の条件に当てはまる場合は登録できない。

- 新規にセッションに登録しようとするオブジェクトのキー名が、ライフサイクル管理外のセッションで既に登録されていた場合
- Add メソッドで指定するライフサイクルレベルと異なるライフサイクルレベルで既に管理されていた場合

(2) セッション情報の操作に関する注意事項

SessionManager によって登録したオブジェクトやマスタデータは、SessionManager 以外で操作しないこと。SessionManager 以外の手段でこれらのデータを操作した場合、マスタデータで管理している内容と実際のセッションに登録されている項目に相違が生じる可能性がある。このようなことによって生じたマスタデータの矛盾には、セッション管理機能は対処しない。従って、SessionManager を通してセッションに登録したデータは、SessionManager を通して操作すること。

◆ セッション情報のライフサイクル単位での一括削除機能

セッション管理機能では、セッションに登録されているマスタデータをライフサイクル単位での一括削除機能を提供する。一括削除機能では、クリア対象のライフサイクルレベルを指定すると、それに包含される全てのライフサイクルレベルのセッション情報を削除する。

一括削除機能の利用例をリスト 2 に示す。まず、Add メソッドによって 4 個のオブジェクトをセッションに登録する。登録されたオブジェクトは、図 2 のようにライフサイクル管理される。その後 Remove メソッドでライフサイクルレベル 1 を指定すると、ライフサイクルレベル 1 に包含される、全てのセッション情報を削除する。つまり、ライフサイクルレベル 1 の object11、ライフサイクルレベル 2 の object12 と object13 の 3 つのオブジェクトが削除される。

```
SessionManager manager = new SessionManager(Session);
// Add(セッションキー名、セッションに格納するオブジェクト、ライフサイクルレベル)
manager.Add("key10", object10, 0);
manager.Add("key11", object11, 1);
manager.Add("key12", object12, 2);
manager.Add("key13", object13, 2);

// Remove(ライフサイクルレベル)
manager.Remove(1);
```

リスト 4 セッション一括削除機能の利用例

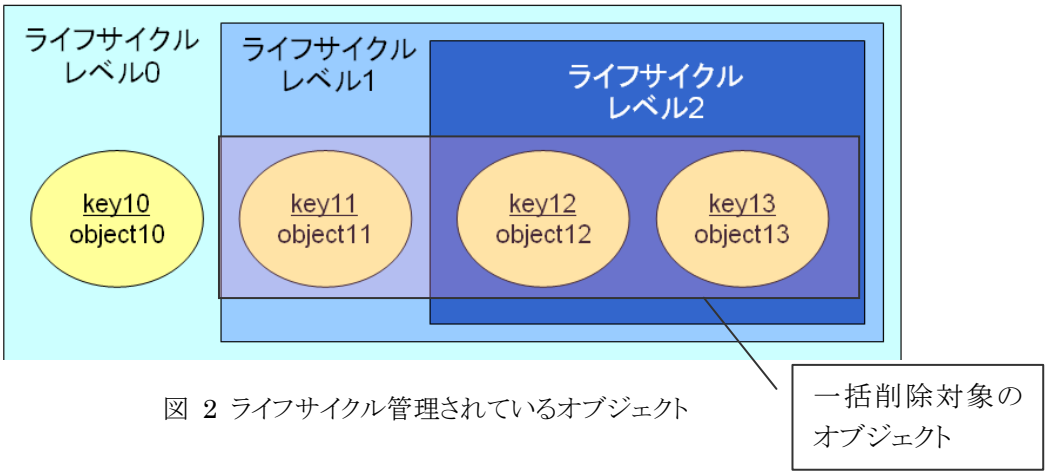


図 2 ライフサイクル管理されているオブジェクト

■ 設計ポイント

ソフトウェアアーキテクトは、ライフサイクルレベル毎のセッション登録情報と、セッションのクリアタイミングについて設計する必要がある。

ライフサイクルレベル毎のセッション登録情報の設計例を表 2 に示す。ここでは、ライフサイクルレベルを業務ごとに定義している。ライフサイクルレベル 0 には、アプリケーション共通で利用する情報を登録する。ライフサイクルレベル 1 には、予約登録・変更・照会などの各ユースケースで利用する情報を登録する。例えば、予約登録情報は、一連の予約登録ユースケースである、予約登録画面から予約登録完了画面、会員メニュー画面に戻るまで保持される。

表 2 ライフサイクルレベル毎のセッション登録情報一覧の例

ライフサイクルレベル	業務	セッション登録情報
レベル 0	-	ログイン情報
レベル 1	予約登録	予約登録情報
	予約変更	予約変更情報
	予約照会	予約照会情報
	-	戻り先画面
	-	エラーメッセージ

セッションのクリアタイミングの設計例を表 3 に示す。ここでは、ユースケースが切り替わる画面遷移のタイミングで、セッション情報をクリアする。具体的に設計するべきことは、どの画面の、どのイベントハンドラで、どのライフサイクルレベルのセッションをクリアするのかという 3 点である。

表 3 セッションのクリアタイミング一覧の例

クリアセッション	クリアタイミング	クリア処理を実装するイベントハンドラ
レベル 0	ログイン画面の初期表示処理	画面#SC0010, PageLoad
	共通エラー画面のトップページへボタン押下処理	画面#SCERR0001, ToTopPageButton_Click
レベル 1	会員メニュー画面の初期表示処理	画面#SC0020, PageLoad

画面遷移図にセッション登録情報とクリアタイミングをマージした例を図 3 に示す。画面遷移図にライフサイクルレベルに関する情報を追記することで、画面とライフサイクルの関係が分かりやすくなる。

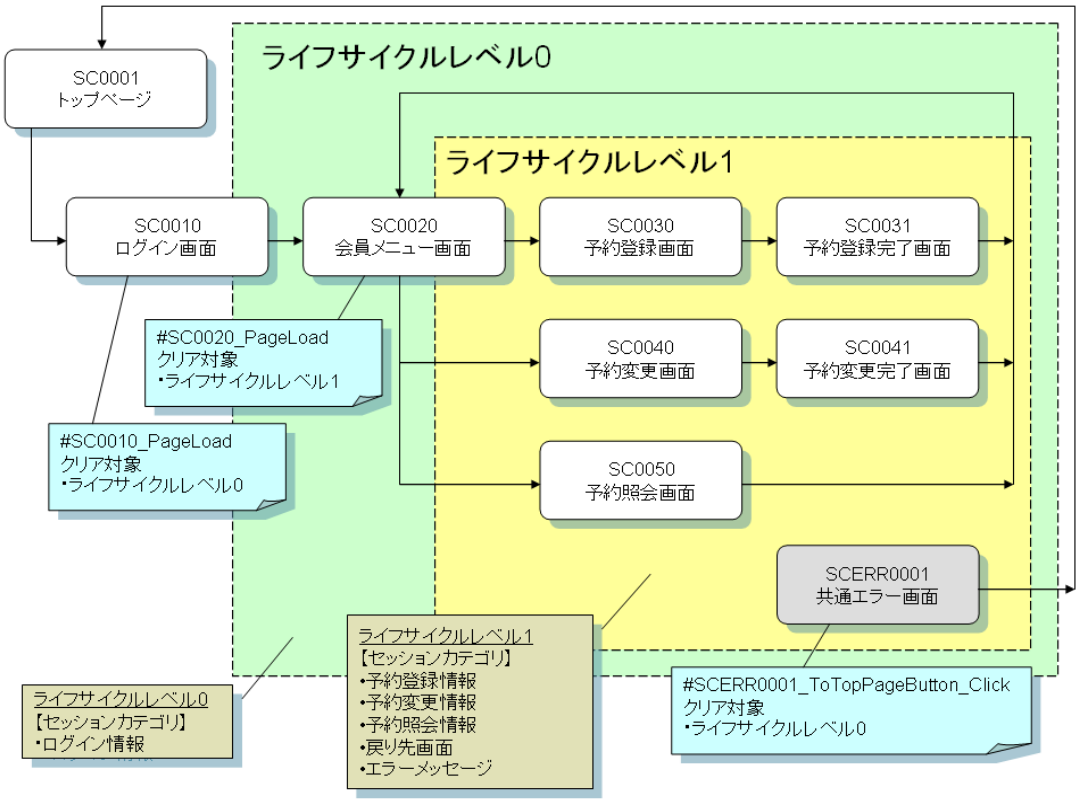


図 3 セッションカテゴリ情報とクリアタイミングを示した画面遷移図

WC-02 SQL 文管理機能

■ 概要

本機能は、設定ファイルに定義した SQL 文を取得する機能を提供する。これにより、SQL 文によってパフォーマンスチューニングしたい場合などに、再ビルドを行わずに SQL 文を変更することができる。

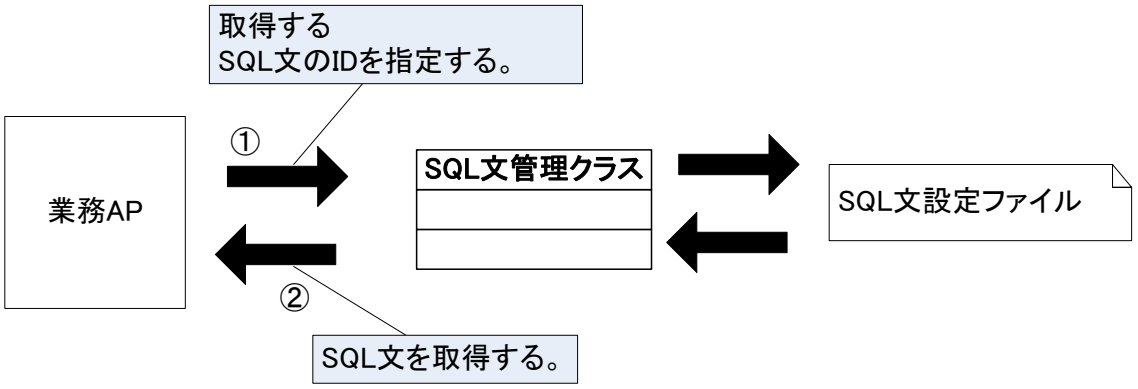


図 1 SQL 文管理機能

業務 AP で利用する SQL 文を設定ファイルに定義し、コードから SQL 文に対応する名前を指定することで、SQL 文を動的に取得することができる機能である。

SQL 文を記述する設定ファイルは、Web 構成ファイル(web.config)に利用する設定ファイルのパスを複数記述することができるため、ユースケースごとに SQL 文設定ファイルを分割することも可能である。

■ 使用方法

◆ Web 構成ファイル

本機能を有効にする場合、Web 構成ファイル(web.config)に SQL 文設定ファイルを定義する。

表 1 Web 構成ファイル

ノード	属性	必須	値
/configuration/configSections/section			複数可
	name		構成要素名。 固定値。以下の値とする。 sqlConfiguration
	type		構成設定の処理を行う構成セクションハンドラクラス名。 固定値。以下の値とする。 TERASOLUNA.Fw.Web.Configuration.Sql.SqlConfigurationSection,TERASOLUNA.Fw.Web
/configuration/sqlConfiguration/files/file			複数可
	path	○	SQL 文設定ファイルのアプリケーションルートからのパス。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  …
  <configSections>
    <section name="sqlConfiguration"
             type="TERASOLUNA.Fw.Web.Configuration.Sql.SqlConfigurationSection,TERASOLUNA.Fw.Web"/>
  </configSections>
  <sqlConfiguration>
    <files>
      <file path="Config¥SqlConfiguration01.config"/>
      <file path="Config¥SqlConfiguration02.config"/>
    </files>
  </sqlConfiguration>
```

リスト 1 Web 構成ファイル記述例

◆ SQL 文設定ファイル

SQL 文設定ファイルにより、SQL 文に対応する ID（SQL ID）と SQL 文を関連付ける。

表 2 SQL 文設定ファイル

ノード	属性	必須	値
/sqlConfiguration	xmlns	○	Namespace 名。 固定値。以下の値とする。 http://www.terasoluna.jp/schema/SqlSchema.xsd
/sqlConfiguration/sql			複数可。
	name	○	SQLID。全ての SQL 文設定ファイルの中で一意になる設定をする。重複不可。

/sqlConfiguration/sql の値として、SQL 文を設定する。記入方法は、XML で利用可能な文字をすべて記述することができる用に、CDATA セクションで囲み記入する。(”![CDATA[sql 文]”])

```
<?xml version="1.0" encoding="utf-8" ?>
<sqlConfiguration xmlns="http://www.terasoluna.jp/schema/SqlSchema.xsd">
  <sql name="selectCust"><![CDATA[SELECT 1 FROM TABLE]]></sql>
  <sql name="updateCust"><![CDATA[UPDATE TABLE SET COLUMN = 2]]></sql>
  <sql name="insertCust"><![CDATA[INSERT INTO TABLE VALUES (3)]]></sql>
</sqlConfiguration>
```

リスト 2 SQL 文設定ファイル記述例

◆ 実装方法

SQL 文設定ファイルから SQL 文を取得する場合は、SqlConfiguration クラスの GetSql メソッドに SQL ID を指定して実行する。

```
string sql = SqlConfiguration.GetSql("selectCust");
```

リスト 3 実装例