↑ Top

目次

改訂情報

変更年月日	変更内容
2015-12-01	初版
2016-08-01	第2版　下記を変更しました ■ フロー要素作成におけるコードに含まれるコンストラクタの修飾子をprotectedからpublicに修正 ■ フロー要素にプロパティを追加するコード例に不足していたimport文を追加 ■ 「*フロートリガ*」を追加。

はじめに

本書の目的

本書では IM-LogicDesigner におけるそれぞれの機能を拡張する仕組の詳細について説明します。

説明範囲は以下のとおりです。

- フロー要素
- マッピング
- Expression Language

対象読者

本書では次の利用者を対象としています。

- intra-mart Accel Platform を理解している

対象開発モデル

本書では以下の開発モデルを対象としています。

- JavaEE開発モデル

サンプルコードについて

本書に掲載されているサンプルコードは可読性を重視しており、性能面や保守性といった観点において必ずしも適切な実装ではありません。
開発においてサンプルコードを参考にされる場合には、上記について十分に注意してください。

本書の構成

- *拡張パッケージ*

  IM-LogicDesigner 機能を拡張するには、必ず拡張機能を含んだパッケージを指定する必要があります。
  指定されたパッケージ配下に存在する全ての拡張機能は、intra-mart Accel Platform 起動時に自動的に検出され読み込まれます。
  その為、本書に掲載されている全ての拡張機能を実装する際には必ずこのパッケージ指定を行う必要があります。

- *フロー要素*

  IM-LogicDesigner で利用するフロー要素以外の独自のフロー要素、カテゴリを追加したい開発者向け

です。
この追加を行うことで、IM-LogicDesigner 上に様々な独自処理を追加することができるようになります。

- *マッピング関数*

  ロジックフローにおける、データの受け渡しを行うマッピング機能に、独自の関数を追加したい開発者向けです。
  この追加を行うことで、マッピング時に、任意の処理関数を追加することができるようになります。

- *EL関数*

  IM-LogicDesigner では、Expression Languageにより、分岐条件、繰り返し条件を指定する事が可能です。
  このExpression Languageに対し、独自の関数を追加する方法を解説します。

- *データ変換*

  マッピング時に、それぞれデータ型の違う値をマッピングした場合には、データ型の変換が行われます。
  このデータ型の変換処理を独自に追加したい開発者向けです。

拡張パッケージ

本章では、IM-LogicDesigner における全ての拡張機能を実装する為に必要なパッケージ指定方法を解説します。
指定されたパッケージは、そのパッケージ配下のサブパッケージを含めて拡張機能を検索します。検索対象となるパッケージの範囲が広い場合には、起動時の解析に時間がかかる可能性があります。
指定するパッケージは最小限の範囲を指定してください。

パッケージ指定クラスの作成

パッケージの指定を行うには、パッケージを提供するためのクラスを実装する必要があります。
パッケージ指定を行うクラスは、必ず `jp.co.intra_mart.system.logic.factory.ElementScanPackageFactory` インタフェースを実装する必要があります。
本書では、拡張パッケージとして **org.example.logicdesigner** パッケージを指定します。
以降の章で解説するすべての機能は **org.example.logicdesigner** パッケージ配下に作成します。

パッケージを指定するためのMyPackageFactoryクラスを作成します。

```
package org.example.logicdesigner;

import java.util.Arrays;
import java.util.Collection;

import jp.co.intra_mart.system.logic.factory.ElementScanPackageFactory;

public class MyPackageFactory implements ElementScanPackageFactory {

    @Override
    public Collection<String> getTargetPackages() {
        return Arrays.asList("org.example.logicdesigner");
    }
}
```

パッケージ指定クラスの登録

作成したMyPackageFactoryクラスは、ServiceLoaderクラスを利用して読み込まれるサービスプロバイダとして扱います。
その為、ServiceLoaderとして読み込みを行う為のプロバイダ構成ファイルを配置します。
プロバイダ構成ファイルはクラスパス上の `/META-INF/services/jp.co.intra_mart.system.logic.factory.ElementScanPackageFactory` ファイルとなります。
intra-mart e Builder for Accel Platform を利用する場合には、プロジェクト配下より

src/main/resources/META-INF/services ディレクトリを作成した後、

jp.co.intra_mart.system.logic.factory.ElementScanPackageFactory ファイルを作成してください。

プロバイダ構成ファイルには、作成したパッケージ指定クラスのFQDNを指定します。

```
org.example.logicdesigner.MyPackageFactory
```

フロー要素

本章では、フロー実行時に呼び出されるフロー要素の作成方法を解説します。
フロー要素を追加するには、以下のクラスを作成する必要があります。

- カテゴリ
- フロー要素
- メタデータ

カテゴリは一度作成を行ったら、再利用することが可能です。

カテゴリの作成

はじめに、フロー要素が所属するカテゴリの作成を行います。
カテゴリは、ロジックフローのデザイナ画面左部に表示されるパレットで利用されます。
カテゴリは全てJavaクラスで作成する必要があります。

カテゴリは `jp.co.intra_mart.foundation.logic.element.category.ElementCategory` インタフェースを実装したカテゴリクラスを作成してください。

```
package org.example.logicdesigner.element;

import jp.co.intra_mart.foundation.logic.element.category.ElementCategory;

public class MyCategory implements ElementCategory {

   @Override
   public String getCategoryId() {
      return "my_category";
   }

   @Override
   public String getDisplayName() {
      return "サンプルカテゴリ";
   }

   @Override
   public int getSortNumber() {
      return 100;
   }
}
```

- **getCategoryId** には、カテゴリのIDとなる文字列を返却するよう実装します。
  getCategoryIdには、**im_** で始まる文字列は利用できません。

- **getDisplayName** は、画面上に表示されるカテゴリ名を返却するよう実装します。
  多言語化を行う場合には、MessageManager API等を利用して利用者のロケールに沿ったカテゴリ名を返却するよう実装を行う必要があります。
- **getSortNumber** は、カテゴリを表示する際に利用するソート順となります。getSortNumberの値が小さい数ほど先頭に表示されるようになります。
  標準機能のソート番号は、1以降の連番を利用しています。

フロー要素とメタデータの作成

フロー要素の引数、戻り値の作成

フロー要素は、ロジックフロー実行時に指定のメソッドが呼び出されます。
指定のメソッドには任意の引数、戻り値を用意することが可能です。
引数、戻り値に利用可能なデータ型は以下の通りです。

- プリミティブ型
- java.lang.String
- java.lang.Boolean
- java.lang.Byte
- java.lang.Character
- java.lang.Short
- java.lang.Integer
- java.lang.Long
- java.lang.Float
- java.lang.Double
- java.math.BigDecimal
- java.math.BigInteger
- java.util.Calendar
- java.util.Date
- java.util.Locale
- java.util.TimeZone
- jp.co.intra_mart.foundation.i18n.datetime.DateTime
- jp.co.intra_mart.foundation.i18n.datetime.Duration
- java.sql.Date
- java.sql.Timestamp
- jp.co.intra_mart.foundation.logic.data.basic.ByteArrayBinary
- jp.co.intra_mart.foundation.logic.data.basic.InputStreamBinary
- jp.co.intra_mart.foundation.service.client.file.PublicStorage

- jp.co.intra_mart.foundation.service.client.file.SessionScopeStorage
- java.util.Mapを実装するクラス java.util.HashMap等
- java.lang.Object

上記のデータ型を内包する java.util.Collection インタフェース、または、java.util.List インタフェースの実装クラス(ArrayList, LinkedList等)を利用することが可能です。
Collection、Listを利用した場合には、@TypeHint アノテーションを付与し、Collection、Listが内包する型を指定する必要があります。
@TypeHint アノテーションは、読み取り用メソッドに付与してください。

ここでは、引数、戻り値となるクラスを作成します。作成するクラスが持つプロパティを利用するには、対象のプロパティに対するgetter、および、setterを持つ必要があります。
実装するgetter、および、setterはJavaBeansの規約に沿って実装を行ってください。

```
package org.example.logicdesigner.element;

import java.util.Collection;

import jp.co.intra_mart.foundation.logic.annotation.TypeHint;

public class MyParameter {

    private String stringParameter;
    private boolean booleanParameter;
    private String[] stringArrayParameter;
    private Collection<Integer> integerListParameter;

    public String getStringParameter() {
        return stringParameter;
    }

    public void setStringParameter(String stringParameter) {
        this.stringParameter = stringParameter;
    }

    public boolean isBooleanParameter() {
        return booleanParameter;
    }

    public void setBooleanParameter(boolean booleanParameter) {
        this.booleanParameter = booleanParameter;
    }

    public String[] getStringArrayParameter() {
        return stringArrayParameter;
    }

    public void setStringArrayParameter(String[] stringArrayParameter) {
        this.stringArrayParameter = stringArrayParameter;
    }

    @TypeHint(Integer.class)
    public Collection<Integer> getIntegerListParameter() {
        return integerListParameter;
    }

    public void setIntegerListParameter(Collection<Integer> integerListParameter) {
        this.integerListParameter = integerListParameter;
    }
}
```

```
package org.example.logicdesigner.element;

public class MyResult {

    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

フロー要素、メタデータクラスの作成

次に、メインのクラスとなるフロー要素クラスを実装していきます。
フロー要素を作成する前に、フロー要素に対応したメタデータクラスを作成します。
メタデータクラスは、jp.co.intra_mart.foundation.logic.element.metadata.FlowElementMetadata クラスを継承して作成してください。

```
package org.example.logicdesigner.element;

import jp.co.intra_mart.foundation.logic.element.metadata.FlowElementMetadata;

public class MyTaskMetadata extends FlowElementMetadata {

    public MyTaskMetadata() {
        super(null);
    }

    @Override
    public String getElementName() {
        return "サンプルタスク";
    }
}
```

- **getElementName** メソッドは、パレット上に表示される名前となります。
  多言語化を行う場合には、MessageManager API等を利用して利用者のロケールに沿った要素名を返却するよう実装を行う必要があります。
- コンストラクタ内で呼び出しを行っている super(null)部分は暫定的に記述しています。この部分はフロー要素クラスを作成後に変更を行います。

メタデータクラスを作成したら、フロー要素クラスを作成します。
フロー要素は、jp.co.intra_mart.foundation.logic.element.Task クラスを継承して作成してください。
Taskクラスには、3つの型パラメータを指定します。メタデータクラス、Taskに受け渡される引数の型、Taskが返却する戻り値の型を指定してください。

ここでは前項で作成したパラメータ用の型と返却用の型を指定します。
また、フロー要素には、jp.co.intra_mart.foundation.logic.annotation.LogicFlowElement アノテーションを付与する必要があります。LogicFlowElementアノテーションが付与されたクラスは起動時に自動的に読み込まれます。

```
package org.example.logicdesigner.element;

import jp.co.intra_mart.foundation.logic.element.ElementContext;
import jp.co.intra_mart.foundation.logic.element.Task;
import jp.co.intra_mart.foundation.logic.exception.FlowExecutionException;

@LogicFlowElement(id="my_task", category=MyCategory.class, index=100)
public class MyTask extends Task<MyTaskMetadata, MyParameter, MyResult> {

    public MyTask(ElementContext context) {
        super(context);
    }

    @Override
    public MyResult execute(MyParameter parameter) throws FlowExecutionException {

        System.out.println(parameter.getStringParameter());

        MyResult result = new MyResult();
        result.setMessage("hello world.");
        return result;
    }
}
```

- executeメソッドがフロー要素実行時に呼び出されるメソッドとなります。任意の処理を記述してください。
- LogicFlowElementアノテーションには、そのフロー要素のID(id)、所属するカテゴリ(category)、パレットに表示する際に利用されるソート番号(index)を指定してください。
- **im_** で始まるIDを指定することは出来ません。前項で作成したMyCategoryクラスをカテゴリとして指定しています。
- コンストラクタでは、ElementContext を引数に受け取ります。ElementContext はフロー実行時の情報が格納されています。
- 後述する FlowElementCloser を ElementContext に登録することにより、フロー実行後の後処理を行うプログラムを登録することが可能です。
- フロー要素はロジックフロー実行時に都度インスタンスが生成されます。
  但し、繰り返し等で同一のフロー要素が呼び出される場合にはそのインスタンスは再利用されます。

次に、メタデータクラスを修正します。
メタデータクラスのコンストラクタ内で、親クラスのコンストラクタを呼び出している部分に、作成したフロー要素クラスを受け渡します。
メタデータクラスは、フロー要素のメタ情報を扱います。フロー要素の入力値、出力値が持つデータ型や、後述するプロパティに関する情報を提供する役割です。
親コンストラクタの引数にフロー要素クラスを受け渡すことにより、自動的にフロー要素の入力値、出力値が持つ

データ型を解析し、メタ情報として内包します。
独自に入力値、出力値等のメタ情報を扱いたい場合には、FlowElementMetadata の持つ各メソッドをオーバーライドしてください。

```
package org.example.logicdesigner.element;

import jp.co.intra_mart.foundation.logic.element.metadata.FlowElementMetadata;

public class MyTaskMetadata extends FlowElementMetadata {

    public MyTaskMetadata() {
        super(MyTask.class);
    }

    @Override
    public String getElementName() {
        return "サンプルタスク";
    }
}
```

引数、戻り値、メタデータ、フロー要素の作成を行いました。
作成したクラスを intra-mart Accel Platform 上に配置することにより、作成したフロー要素が認識され利用できるようになります。

フロー要素にプロパティを追加する

フロー要素の引数、戻り値はロジックフローのマッピングにて利用されますが、引数、戻り値以外に、事前に決められた値を設定しておくためのプロパティ機構が存在します。
プロパティに利用可能なデータ型は以下の通りです。

- プリミティブ型
- java.lang.String
- java.lang.Boolean
- java.lang.Character
- java.lang.Short
- java.lang.Integer
- java.lang.Long
- java.lang.Float
- java.lang.Double
- java.math.BigDecimal
- java.math.BigInteger
- java.util.Date
- java.sql.Date
- java.sql.Timestamp

- java.lang.Enum

プロパティを追加するには、フロー要素クラスにプロパティとして利用するフィールド、および、getter、setterを追加します。
今回は、前項で作成した MyTask に customProperty という名のプロパティを追加します。

```
package org.example.logicdesigner.element;

import jp.co.intra_mart.foundation.logic.element.ElementContext;
import jp.co.intra_mart.foundation.logic.element.Task;
import jp.co.intra_mart.foundation.logic.exception.FlowExecutionException;

@LogicFlowElement(id="my_task", category=MyCategory.class, index=100)
public class MyTask extends Task<MyTaskMetadata, MyParameter, MyResult> {

    private String customProperty;

    public MyTask(ElementContext context) {
        super(context);
    }

    public String getCustomProperty() {
        return customProperty;
    }

    public void setCustomProperty(String customProperty) {
        this.customProperty = customProperty;
    }

    @Override
    public MyResult execute(MyParameter parameter) throws FlowExecutionException {

        System.out.println(parameter.getStringParameter());

        MyResult result = new MyResult();
        result.setMessage("hello world.");
        return result;
    }
}
```

プロパティを追加したら、そのままデザイナの設定画面から値の設定を行うことが可能となります。
デザイナの設定画面では、**customProperty** として項目名が表示されますが、メタデータクラスを変更することにより任意の表示名に変更することが可能です。

```
package org.example.logicdesigner.element;

import jp.co.intra_mart.foundation.logic.element.metadata.FlowElementMetadata;
import jp.co.intra_mart.foundation.logic.element.metadata.ElementProperty;

public class MyTaskMetadata extends FlowElementMetadata {

    public MyTaskMetadata() {
        super(MyTask.class);
    }

    @Override
    public String getElementName() {
        return "サンプルタスク";
    }

    @Override
    protected ElementProperty decorateElementProperty(ElementProperty elementProperty) {
        if("customProperty".equals(elementProperty.getPropertyName())) {
            elementProperty.setDefaultValue("hello world.");
            elementProperty.setLabelKey("MYTASK.CUSTOMPROPERTY.LABEL.KEY");
        }
        return elementProperty;
    }
}
```

- **decorateElementProperty** メソッドをオーバーライドし、プロパティ項目のカスタマイズが可能です。
- **setDefaultValue** により、デフォルト値の指定を行っています。
- **setLabelKey** にはプロパティの表示名にあたる多言語化リソースのキーを指定します。
- **setType** メソッドを利用することにより画面上変更することが可能です。プロパティにboolean型を指定し、**flag** タイプを指定することによりチェックボックスとして扱えます。

フロー要素に後処理を追加する

ロジックフロー実行後に、任意の処理を呼び出すことが可能です。
これは、フロー要素内で使用したリソースの開放等を行う際に利用します。
後処理は、`jp.co.intra_mart.foundation.logic.element.FlowElementCloser` インタフェースを実装する必要があります。
本項では、作成した `MyTask` クラスに直接 `FlowElementCloser` インタフェースを実装しています。
フロー要素のコンストラクタに受け渡される `ElementContext` に対し、**addFlowElementCloser** メソッドで `FlowElementCloser` を登録します。

```
package org.example.logicdesigner.element;

import jp.co.intra_mart.foundation.logic.annotation.LogicFlowElement;
import jp.co.intra_mart.foundation.logic.element.ElementContext;
import jp.co.intra_mart.foundation.logic.element.FlowElementCloser;
import jp.co.intra_mart.foundation.logic.element.Task;
import jp.co.intra_mart.foundation.logic.exception.FlowExecutionException;

@LogicFlowElement(id = "my_task", category = MyCategory.class, index = 100)
public class MyTask extends Task<MyTaskMetadata, MyParameter, MyResult> implements
FlowElementCloser {

    private String customProperty;

    public MyTask(ElementContext context) {
        super(context);
        context.addFlowElementCloser(this);
    }

    @Override
    public MyResult execute(MyParameter parameter) throws FlowExecutionException {

        System.out.println(parameter.getStringParameter());

        MyResult result = new MyResult();
        result.setMessage("hello world.");
        return result;
    }

    @Override
    public void close() {
        System.out.println("closed.");
    }

    public String getCustomProperty() {
        return customProperty;
    }

    public void setCustomProperty(String customProperty) {
        this.customProperty = customProperty;
    }
}
```

- コンストラクタ内で **addFlowElementCloser** メソッドを呼び出し自身のインスタンスを `FlowElementCloser` として登録しています。
- `FlowElementCloser` インタフェースの持つ **close** メソッドを実装しています。

マッピング関数

本章では、IM-LogicDesigner の持つマッピング機能で利用可能なマッピング関数の定義方法について解説します。

マッピング関数の引数と戻り値

マッピング関数で利用可能な引数と戻り値は以下の通りです。

- プリミティブ型
- java.lang.String
- java.lang.Boolean
- java.lang.Byte
- java.lang.Character
- java.lang.Short
- java.lang.Integer
- java.lang.Long
- java.lang.Float
- java.lang.Double
- java.math.BigDecimal
- java.math.BigInteger
- java.util.Calendar
- java.util.Date
- java.util.Locale
- java.util.TimeZone
- jp.co.intra_mart.foundation.i18n.datetime.DateTime
- jp.co.intra_mart.foundation.i18n.datetime.Duration
- java.sql.Date
- java.sql.Timestamp
- jp.co.intra_mart.foundation.logic.data.basic.ByteArrayBinary
- jp.co.intra_mart.foundation.logic.data.basic.InputStreamBinary
- jp.co.intra_mart.foundation.service.client.file.PublicStorage
- jp.co.intra_mart.foundation.service.client.file.SessionScopeStorage

- java.util.Mapを実装するクラス java.util.HashMap等

上記のデータ型を内包する java.util.Collection インタフェースまたは、java.util.List インタフェースの実装クラス(ArrayList, LinkedList等)を利用することが可能です。

マッピング関数の引数、戻り値と関連付けられているデータ型が一致していない場合には自動的にデータ型の変換が行われた後、マッピング関数が呼びだされます。
マッピング関数の引数、戻り値が配列/リスト形式であり、関連付けられているデータ型が配列/リスト形式でない場合には自動的に変換が行われます。
変換の詳細に関しては IM-LogicDesigner 仕様書、マッピングの章を参照してください。

マッピング関数カテゴリの作成

マッピング関数には、その関数の所属するカテゴリが存在します。
カテゴリを作成するには、
jp.co.intra_mart.foundation.logic.data.mapping.function.MappingFunctionCategory インタフェースを実装したカテゴリクラスを作成する必要があります。

```
package org.example.logicdesigner.data;

import jp.co.intra_mart.foundation.logic.data.mapping.function.MappingFunctionCategory;

public class MyFunctionCategory implements MappingFunctionCategory {

   @Override
   public String getCategoryId() {
      return "my_func_category";
   }

   @Override
   public String getDisplayName() {
      return "サンプルカテゴリ";
   }

   @Override
   public int getSortNumber() {
      return 100;
   }
}
```

- **getCategoryId** では、一意となるカテゴリIDを返却するよう実装してください。
  getCategoryIdには、**im_** で始まるIDは利用できません。
- **getDisplayName** は、画面上に表示されるカテゴリ名として利用されます。
  多言語化を行う場合には、MessageManager API等を利用して利用者のロケールに沿ったカテゴリ名を返却するよう実装を行う必要があります。
- **getSortNumber** は、画面上に表示する際に利用されるソート番号です。標準で提供されるカテゴリは10, 20, 30...と連番で指定が行われています。

マッピング関数の作成

マッピング関数を作成するには、jp.co.intra_mart.foundation.logic.data.mapping.Function インタフェースを実装する必要があります。
実装を容易にするための jp.co.intra_mart.system.logic.data.mapping.function.AbstractFunction クラスが用意されています。
マッピング関数として起動時に検出対象となるよう、
jp.co.intra_mart.foundation.logic.annotation.MappingFunction アノテーションをクラスに指定してください。
作成したマッピング関数は、アノテーションを付与することにより起動時に自動的に検出、登録が行われます。その為設定ファイルは不要です。

マッピング関数は、フロー定義に含まれるマッピング毎にインスタンスが作成されます。マッピング関数はスレッドセーフとなるよう実装する必要があります。

```
package org.example.logicdesigner.data;

import java.math.BigDecimal;

import jp.co.intra_mart.foundation.logic.annotation.MappingFunction;
import jp.co.intra_mart.foundation.logic.data.mapping.MappingContext;
import jp.co.intra_mart.foundation.logic.exception.FunctionInvocationException;
import jp.co.intra_mart.system.logic.data.mapping.StandardArgumentType;
import jp.co.intra_mart.system.logic.data.mapping.StandardReturnType;
import jp.co.intra_mart.system.logic.data.mapping.function.AbstractFunction;

@MappingFunction(category = MyFunctionCategory.class, index = 100)
public class MyFunction extends AbstractFunction {

   private static final long serialVersionUID = -1;

   @Override
   public String getId() {
      return "my_tax_function";
   }

   @Override
   public String getName() {
      return "tax";
   }

   @Override
   protected void initialize() {
      addArgumentType(StandardArgumentType.BIGDECIMAL);
      setReturnType(StandardReturnType.BIGDECIMAL);
   }

   @Override
   public Object execute(MappingContext context, Object... arguments) throws
FunctionInvocationException {
      BigDecimal argument = (BigDecimal) arguments[0];
      return argument.multiply(new BigDecimal("1.08"));
   }
}
```

- `@MappingFunction` アノテーションでは、カテゴリと、カテゴリ内で利用されるソート番号を指定しています。
- **getId** には、一意となるマッピング関数のIDを返却するよう実装してください。
  getIdには、**im_** で始まるIDは利用できません。
- **getName** では、画面上に表示する際に利用されるマッピング関数名を返却するよう実装します。関数名は全て英数字を指定してください。
- **initialize** は、マッピング関数初期化時に呼び出されます。ここでは、引数および戻り値のデータ型を指定してください。
- **execute** は実際にマッピング関数が呼び出された際に実行されます。

- **execute** に渡される引数 MappingContext には、フロー実行中の変数等が格納されています。

EL関数

IM-LogicDesigner では、分岐条件、繰り返し条件等でExpression Language(EL)を利用します。
本章では、EL内で独自の関数を利用できるようEL関数の追加を行う方法を解説します。

EL関数の追加

EL関数を追加するには、public static 修飾子を持つメソッドを実装する必要があります。
また、起動時に自動的に検出を行う為に、jp.co.intra_mart.foundation.logic.annotation.ProvideELFunction アノテーションをクラスに付与します。
EL関数として追加するメソッドには、jp.co.intra_mart.foundation.logic.annotation.ELFunction アノテーションを付与してください。
対象となるメソッドのメソッド名がEL関数名となります。

```
package org.example.logicdesigner.el;

import java.io.UnsupportedEncodingException;
import java.net.URLDecoder;
import java.net.URLEncoder;
import java.nio.charset.StandardCharsets;

import jp.co.intra_mart.foundation.logic.annotation.ELFunction;
import jp.co.intra_mart.foundation.logic.annotation.ProvideELFunction;

@ProvideELFunction
public class MyELFunction {

    @ELFunction(prefix = "my")
    public static String encodeURIComponent(String value) {
        try {
            return URLEncoder.encode(value, StandardCharsets.UTF_8.name());
        } catch (UnsupportedEncodingException ignore) {
            return null;
        }
    }

    @ELFunction(prefix = "my")
    public static String decodeURIComponent(String value) {
        try {
            return URLDecoder.decode(value, StandardCharsets.UTF_8.name());
        } catch (UnsupportedEncodingException ignore) {
            return null;
        }
    }
}
```

- @ELFunction アノテーションには、EL関数のプレフィクスが指定可能です。この例では、${my:encodeURIComponent('
')} のように呼び出すことが可能となります。
- 一つのクラスに複数のEL関数を定義することが可能です。

データ変換

本章では、マッピングの際に利用されるデータ変換処理を追加する方法に関して解説します。
標準機能としてデータ変換機能が提供されています。この機能は標準のデータ変換の仕組みを差し替えるという目的ではなく、複雑な型の変換を行う際のマッピングを簡易化する目的で用意されています。

データ変換処理の作成

データ変換処理を作成するには、jp.co.intra_mart.foundation.logic.data.converter.Converter インタフェースを実装します。
また、起動時にデータ変換処理を検出、登録するため作成したクラスに対して jp.co.intra_mart.foundation.logic.annotation.DataConverter アノテーションを付与してください。
データ変換処理では、配列/リスト形式の値は受け渡されません。事前に配列/リスト形式に含まれる値を抽出し受け渡しが行われます。

```
package org.example.logicdesigner.data;

import jp.co.intra_mart.foundation.logic.annotation.DataConverter;
import jp.co.intra_mart.foundation.logic.data.TypeDefinition;
import jp.co.intra_mart.foundation.logic.data.converter.Converter;
import jp.co.intra_mart.foundation.logic.exception.TypeConvertionException;

import org.example.logicdesigner.element.MyParameter;
import org.example.logicdesigner.element.MyResult;

@DataConverter
public class MyConverter implements Converter {

    @SuppressWarnings("unchecked")
    @Override
    public <T> T convert(Object value, TypeDefinition<?> sourceTypeDefinition, TypeDefinition<?>
targetTypeDefinition) throws TypeConvertionException {
        MyParameter myParameter = (MyParameter) value;
        MyResult result = new MyResult();
        result.setMessage(myParameter.getStringParameter());
        return (T) result;
    }

    @Override
    public boolean isSupportType(TypeDefinition<?> sourceTypeDefinition, TypeDefinition<?>
targetTypeDefinition) {
        return MyParameter.class.equals(sourceTypeDefinition.getType()) &&
MyResult.class.equals(targetTypeDefinition.getType());
    }
}
```

- **isSupportType** メソッドにて、変換対象となるデータ型であるか判定を行うよう実装します。
- **convert** メソッドにて、データ変換処理を実装します。

フロートリガ

本章では、IM-Propagation を利用したフローを実行するトリガを追加する方法を解説します。
IM-Propagation の詳細は「IM-Propagation 仕様書」を参照してください。

トリガを追加するには、以下のクラスを作成する必要があります。

- カテゴリ
- 受信するデータを格納するためのクラス
- データ変換クラス (Decoder)
- データ処理クラス (Procedure)
- マッピング設定

カテゴリは一度作成を行ったら、再利用することが可能です。

> **注意**
>
> ここでは、「IM-Propagationプログラミングガイド」-「データを送る側の実装」を用いて送られてきたデータを受信した際にフローを実行するトリガを例に説明を行います。
> データの送信側の実装が存在しない場合、トリガ設定を行ってもフローは実行されません。

カテゴリの作成

はじめに、トリガが所属するカテゴリの作成を行います。
カテゴリは全てJavaクラスで作成する必要があります。

カテゴリは `jp.co.intra_mart.foundation.logic.trigger.category.TriggerEventCategory` インタフェースを実装したカテゴリクラスを作成してください。

```
package org.example.logicdesigner.trigger;

import jp.co.intra_mart.foundation.logic.trigger.category.TriggerEventCategory

public class MyTriggerCategory implements TriggerEventCategory {

    @Override
    public String getCategoryId() {
        return "my_trigger_category";
    }

    @Override
    public String getDisplayName() {
        return "サンプルカテゴリ";
    }

    @Override
    public int getSortNumber() {
        return 100;
    }
}
```

- **getCategoryId** には、カテゴリのIDとなる文字列を返却するよう実装します。
  getCategoryIdには、**im_** で始まる文字列は利用できません。
- **getDisplayName** は、画面上に表示されるカテゴリ名を返却するよう実装します。
  多言語化を行う場合には、MessageManager API等を利用して利用者のロケールに沿ったカテゴリ名を返却するよう実装を行う必要があります。
- **getSortNumber** は、カテゴリを表示する際に利用するソート順となります。getSortNumberの値が小さい数ほど先頭に表示されるようになります。

受信するデータを格納するためのクラスの作成

受信するデータを格納するためのクラス（以下、「独自モデル」）を用意します。
独自モデルについては、「IM-Propagationプログラミングガイド」-「受信するデータを格納するためのクラスを作成する」を参照してください。

```
package org.example.logicdesigner.trigger;

public class MyTriggerData {

   private String resourceId;

   private String url;

   public String getResourceId() {
      return resorceId;
   }

   public String getUrl() {
      return url;
   }

   public void setResourceId(String resourceId) {
      this.resourceId = resorceId;
   }

   public void setUrl(String url) {
      this.url = url;
   }

}
```

独自モデルのプロパティに利用可能なデータ型は以下の通りです。

- プリミティブ型
- java.lang.String
- java.lang.Boolean
- java.lang.Byte
- java.lang.Character
- java.lang.Short
- java.lang.Integer
- java.lang.Long
- java.lang.Float
- java.lang.Double
- java.math.BigDecimal
- java.math.BigInteger
- java.util.Calendar
- java.util.Date
- java.util.Locale
- java.util.TimeZone

- jp.co.intra_mart.foundation.i18n.datetime.DateTime
- jp.co.intra_mart.foundation.i18n.datetime.Duration
- java.sql.Date
- java.sql.Timestamp
- jp.co.intra_mart.foundation.logic.data.basic.ByteArrayBinary
- jp.co.intra_mart.foundation.logic.data.basic.InputStreamBinary
- jp.co.intra_mart.foundation.service.client.file.PublicStorage
- jp.co.intra_mart.foundation.service.client.file.SessionScopeStorage
- java.util.Mapを実装するクラス java.util.HashMap等
- java.lang.Object

上記のデータ型を内包する java.util.Collection インタフェース、または、java.util.List インタフェースの実装クラス(ArrayList, LinkedList等)を利用することが可能です。
Collection、Listを利用した場合には、そのプロパティのゲッターメソッドに @TypeHint アノテーションを付与し、Collection、Listが内包する型を指定する必要があります。

```
@TypeHint(String.class)
public List<String> getListData() {
   return listData;
}
```

データ変換クラスの作成

次に、「送受信モデル（Generic）」を「独自モデル」に変換する機能を提供するクラス（以下、「データ変換クラス」）を用意します。
データ変換クラスについて、詳しくは「IM-Propagationプログラミングガイド」-「受信側のデータ変換クラス（Decoder）を作成する」を参照してください。

```
package org.example.logicdesigner.trigger;

import jp.co.intra_mart.foundation.propagation.exception.ConvertException;
import jp.co.intra_mart.foundation.propagation.receiver.AbstractDecoder;

public class MyTriggerDecoder extends AbstractDecoder<SampleGenericData, MyTriggerData> {

   @Override
   public MyTriggerData decode(final SampleGenericData generic) throws ConvertException {
      MyTriggerData data = new MyTriggerData();
      data.setResourceId(generic.getResourceId());
      data.setUrl(generic.getUrl());
      return data;
   }

   @Override
   public Class<SampleGenericData> getGenericDataClass() {
      return SampleGenericData.class;
   }

}
```

データ処理クラスの作成

次に、メインのクラスとなるデータ処理クラスを実装していきます。
データ処理クラスは、jp.co.intra_mart.system.logic.trigger.propagation.LogicFlowPropagationTrigger クラスを継承して作成してください。
また、データ処理クラスには jp.co.intra_mart.foundation.logic.annotation.TriggerEvent アノテーションを付与する必要があります。TriggerEventアノテーションが付与されたクラスは起動時にフローを実行するトリガとして読み込まれます。

```
package org.example.logicdesigner.trigger;

import jp.co.intra_mart.foundation.logic.annotation.TriggerEvent;
import jp.co.intra_mart.system.logic.trigger.propagation.LogicFlowPropagationTrigger;

@TriggerEvent(id="my_trigger", category=MyTriggerCategory.class, index=1)
public class MyTrigger extends LogicFlowPropagationTrigger<MyTriggerData> {
}
```

- TriggerEvent アノテーションには、そのトリガのID(id)、所属するカテゴリ(category)、設定画面に表示する際に利用されるソート番号(index)を指定してください。
- 継承した LogicFlowPropagationTrigger クラスで受信データのマッピングおよびトリガ設定で設定されたフローの実行が行われますので、データ処理クラスでフロー実行の処理を実装する必要はありません。

マッピング設定の作成

次に、ここまで作成した「データ変換クラス」「データ処理クラス」を紐付けるためのマッピング設定を作成します。
詳しくは「IM-Propagationプログラミングガイド」-「マッピング設定を作成する」を参照してください。

```
<?xml version="1.0" encoding="UTF-8"?>
<propagation-receivers-config xmlns="http://www.intra-mart.jp/propagation/receivers-config"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.intra-mart.jp/propagation/receivers-config propagation-receivers-
config.xsd">
  <receiver source="jp.co.intra_mart.module_name.function_name.propagation.SenderModuleData"
operationType="DATA_CREATED">
    <decoder class="org.example.logicdesigner.trigger.MyTriggerDecoder"/>
    <procedure class="org.example.logicdesigner.trigger.MyTrigger"/>
  </receiver>
</propagation-receivers-config>
```

付録

独自のビジネスロジックからロジックフローを呼び出す方法

Java開発モデル

Java開発モデルを利用してロジックフローを呼び出すコード例です。

```
package org.example.logicdesigner;

import java.util.HashMap;
import java.util.Map;

import jp.co.intra_mart.foundation.logic.LogicRuntime;
import jp.co.intra_mart.foundation.logic.LogicServiceProvider;
import jp.co.intra_mart.foundation.logic.LogicSession;
import jp.co.intra_mart.foundation.logic.exception.ErrorEndEventException;
import jp.co.intra_mart.foundation.logic.exception.LogicServiceException;
import jp.co.intra_mart.foundation.logic.exception.LogicServiceRuntimeException;

public class CallExample {

    public void call() {

        // IM-LogicDesignerで利用するサービス群を管理するインスタンスを取得します。
        LogicServiceProvider logicServiceProvider = LogicServiceProvider.getInstance();
        // ロジックフロー実行用ランタイムを取得します。
        LogicRuntime runtime = logicServiceProvider.getLogicRuntime();
        try {
            // ロジックフロー実行用セッションを取得します。
            LogicSession session = runtime.createSession("my-flow");
            // フローに受け渡すパラメータを作成します。
            Map<String, Object> input = new HashMap<>();
            // ロジックフローを実行します。
            Map<String, Object> output = (Map<String, Object>) session.execute(input);
            // 実行結果の確認
            System.out.println(output);
        } catch (LogicServiceRuntimeException e) {
            // リカバリ不能な例外の場合にはLogicServiceRuntimeExceptionが通知されます。
            e.printStackTrace();
        } catch (ErrorEndEventException e) {
            // 明示的にエラー終了による終了を行った場合にはErrorEndEventExceptionが通知されます。
            e.printStackTrace();
        } catch (LogicServiceException e) {
            // ロジックフロー実行時に何らかの例外が発生した場合にはLogicServiceExceptionが通知されます。
            e.printStackTrace();
        }
    }
}
```

スクリプト開発モデルを利用してロジックフローを呼び出すコード例です。

```
function call() {
  // ロジックフローに受け渡すパラメータを作成します。
  var input = {};
  // ロジックフローを呼び出します。
  var result = LogicFlowExecutor.execute('my-flow', input);
  if (result.error) {
    // 何らかのエラーが発生しました。
    Debug.console(result);
  }
  var output = result.data;
  Debug.console(output);
}
```