intra-mart

Copyright © 2014 NTT DATA INTRAMART CORPORATION

↑Top

# 目次


- 1. 改訂情報
- 2. はじめに
  - 本書の目的
  - 対象読者
  - 対象開発モデル
  - サンプルコードについて
  - 本書の構成
- 3. 開発手順
  - 全体の流れ
  - 依存関係の設定
  - データを送る
  - データを受け取る
- 4. データを送る側の実装
  - 用意する資材
  - 送信するデータを格納するためのクラスを作成する
  - 送受信モデル（Generic）を作成する
  - 送信側のデータ変換クラス（Encoder）を作成する
  - マッピング設定を作成する
  - 処理結果を格納するクラスを作成する
  - データを送信する
  - 受信側へ情報を公開する
  - 追加情報
- 5. データを受け取る側の実装
  - 用意する資材
  - 送受信モデル（Generic）を作成する
  - 受信するデータを格納するためのクラスを作成する
  - 受信側のデータ変換クラス（Decoder）を作成する
  - 処理結果を格納するクラスを作成する
  - 受信側のデータ処理クラス（Procedure）を作成する
  - マッピング設定を作成する
  - データを受信する
  - 追加情報
- 6. 付録
  - 復元可能なクラスとは
  - 例外クラス
  - セッションのクローズ漏れを探す

# 改訂情報

| 変更年月日 | 変更内容 |
| --- | --- |
| 2014-05-01 | 初版 |

# はじめに

## 本書の目的

本書では IM-Propagation 機能を使用したトリガ・リスナの開発手順について説明します。

説明範囲は以下の通りです。

- IM-Propagation 機能を使用してデータを送受信するための実装方法
- IM-Propagation 機能を使用する上での注意事項

また、本書と合わせて「IM-Propagation 仕様書 」を参照してください。

## 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 で、各モジュールが管轄するデータの変更や操作の完了を検知して処理を行うプログラム（リスナ）を実装したい開発者
- データの変更や操作の完了を、他のモジュールへ通知するプログラム（トリガ）を実装したい開発者

## 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

- *開発手順*

  IM-Propagation 機能を使用する方法について説明します。

- *データを送る側の実装*

  IM-Propagation 機能を使用してデータを送信するための実装方法について説明します。

- *データを受け取る側の実装*

  IM-Propagation 機能を使用してデータを受信するための実装方法について説明します。

- *付録*

  IM-Propagation 機能を使用する上での補足事項です。

# 開発手順

項目


- 全体の流れ
- 依存関係の設定
- データを送る
- データを受け取る


## 全体の流れ

IM-Propagation を使用したデータの伝搬は、下図のように行われます。

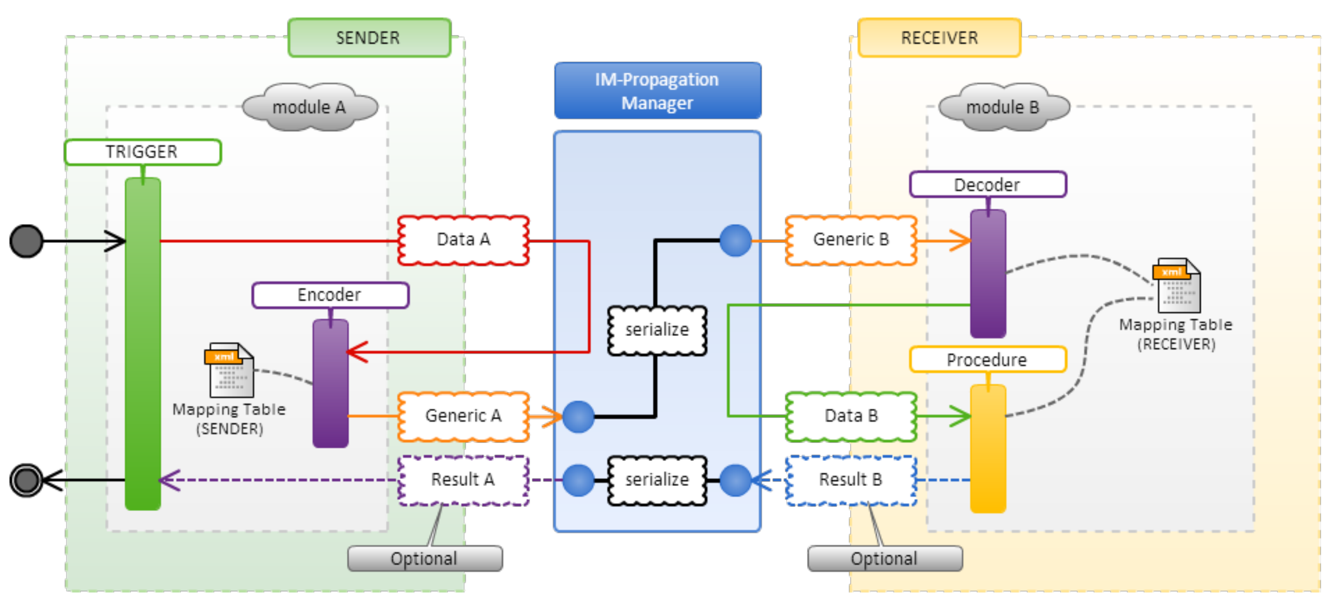


図 データの流れ

処理全体では、おおまかに「送信側」（図中の SENDER）、「受信側」（図中の RECEIVER）、および、それらを接続する「IM-Propagation マネージャ」（図中の IM-Propagation Manager）の3つに分けられます。
そのうち、本書では「送信側」と「受信側」に分けて、実装方法を説明します。

処理の流れについての詳細は、「IM-Propagation 仕様書 」を参照してください。

## 依存関係の設定

データを送受信するためには、送信側・受信側ともに自身のモジュールに対して IM-Propagation モジュールを使用するための設定を行います。
IM-Propagation を使用可能にするためには、モジュールプロジェクト直下に保存されている <module.xml> の「依存関係」に、以下のモジュールを追加します。

| モジュールID | jp.co.intra_mart.im_propagation |
| --- | --- |
| バージョン | 8.0.6 |

コラム

本書で解説している IM-Propagation の説明は、バージョン 8.0.6 時点の動作に基づいて記載されています。
必要に応じて、 8.0.6 以上のバージョンを指定することができます。

データ受信側のモジュールが、データ送信側のモジュールに依存する必要はありません。

## データを送る

送信側の構成は、下図の通りです。

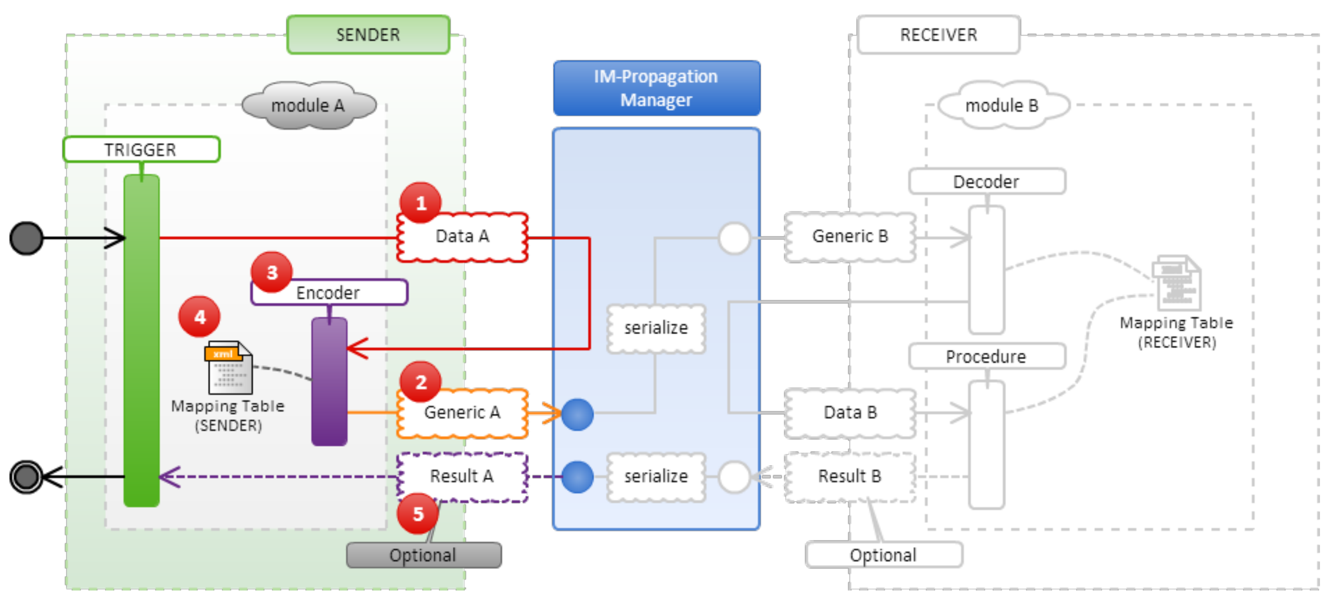


図 送信側の処理

IM-Propagation を使用して、通知やデータを送るために必要な資材は、以下の通りです。

1. 送信処理を行う実装クラス（図中の TRIGGER）
2. 送信するデータを格納するためのクラス（図中の Data A）
3. IM-Propagation にデータを送るためのクラス（図中の Generic A）
4. Data A から Generic A に変換するためのデータ変換クラス（図中の Encoder）
5. 使用するデータ変換クラスを定義するためのマッピング設定（図中の Mapping Table）
6. 処理結果を格納するクラス（図中の Result A）

依存関係を設定した後、上記の資材を用意してください。
各資材を用意する方法についての詳細は、「*データを送る側の実装*」を参照してください。

## データを受け取る

受信側の構成は、下図の通りです。

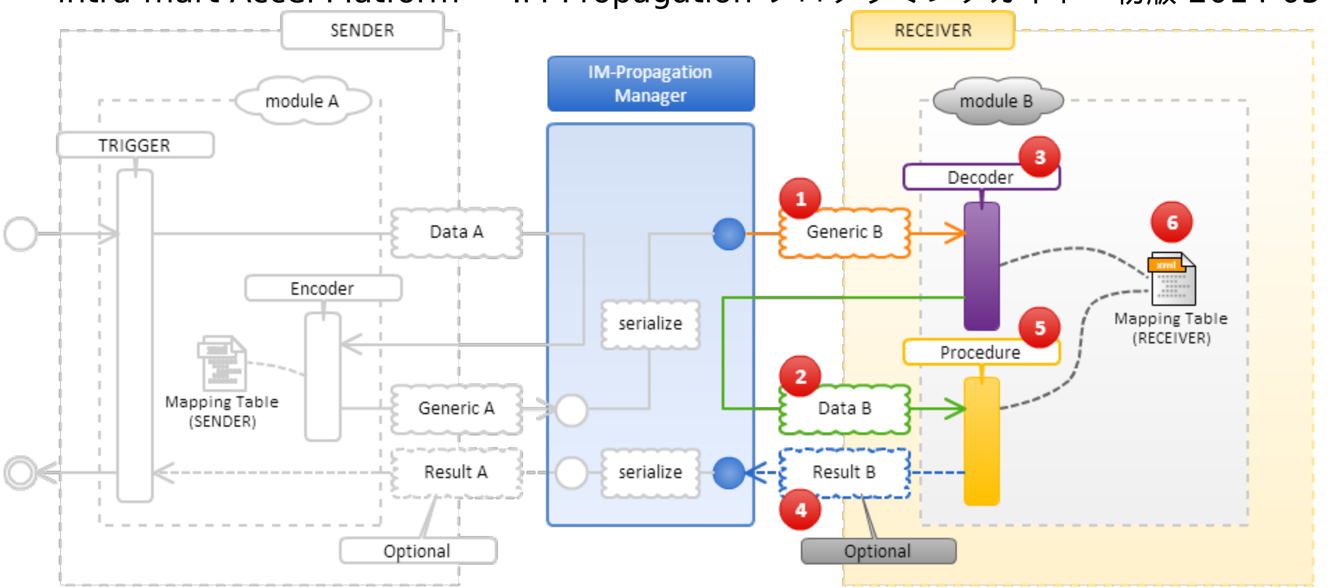


図 受信側の処理

IM-Propagation を使用して送られた通知やデータを、送信側から受け取るために必要な資材は、以下の通りです。

1. IM-Propagation からデータを受け取るためのクラス（図中の Generic B）
2. 受信するデータを格納するためのクラス（図中の Data B）
3. Generic B から Data B に変換するためのデータ変換クラス（図中の Decoder）
4. 処理結果を格納するクラス（図中の Result B）
5. Data B を受け取り処理するためのデータ処理クラス（図中の Procedure）
6. 使用するデータ変換クラス、データ処理クラスを定義するためのマッピング設定（図中の Mapping Table）

依存関係を設定した後、上記の資材を用意してください。
各資材を用意する方法の詳細は、「*データを受け取る側の実装*」を参照してください。

# データを送る側の実装

項目


- 用意する資材
- 送信するデータを格納するためのクラスを作成する
- 送受信モデル（Generic）を作成する
- 送信側のデータ変換クラス（Encoder）を作成する
- マッピング設定を作成する
- 処理結果を格納するクラスを作成する
- データを送信する
- 受信側へ情報を公開する
- 追加情報
  - 「データ変換クラス」の初期パラメータを設定する
  - クロージャ対応のメソッドを使用してデータを送信する
  - セッション管理を使用せずにデータを送信する
  - 受信側から渡された処理結果を取得する


## 用意する資材

送信側で用意する資材は、以下の通りです。

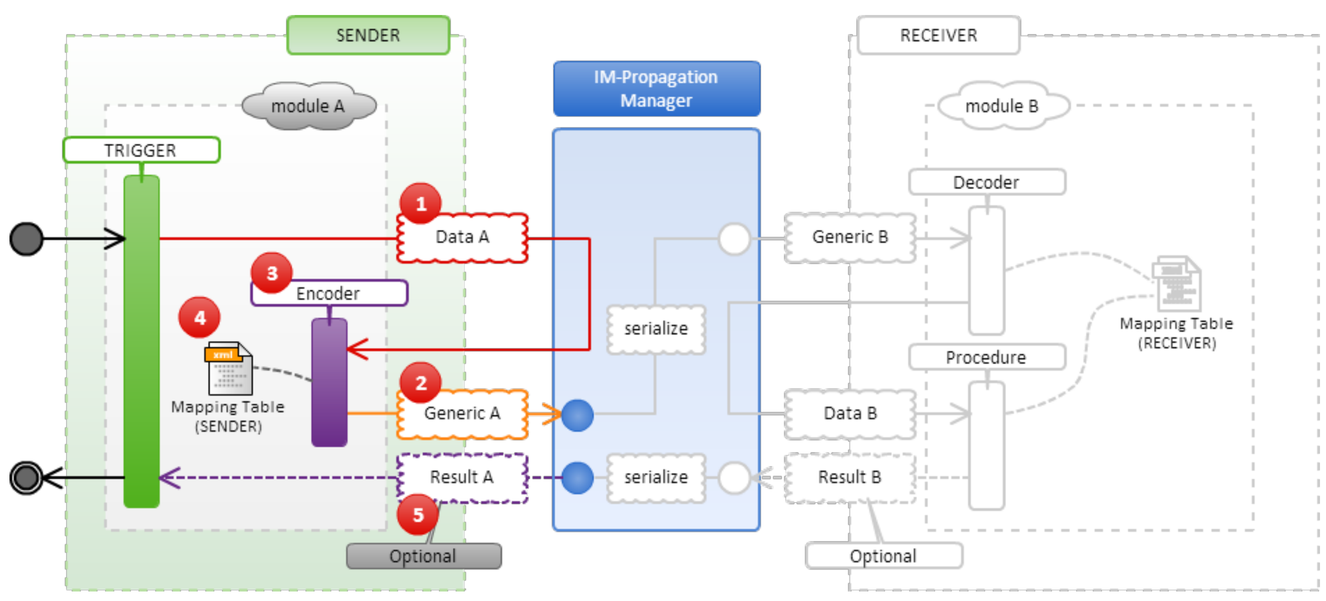


図 送信側の処理

表 用意する資材一覧

| 番号 | 資材名 | 準備必須 | 解説 |
|---|---|---|---|
| 1 | 送信するデータを格納するためのクラス（独自モデル）（図中の Data A） | 必須 | *送信するデータを格納するためのクラスを作成する* |
| 2 | 送受信モデル（Generic）（図中の Generic A） | 必須 | *送受信モデル（Generic）を作成する* |
| 3 | データ変換クラス（図中の Encoder） | 必須 | *送信側のデータ変換クラス（Encoder）を作成する* |
| 4 | マッピング設定（図中の Mapping Table） | 必須 | *マッピング設定を作成する* |

| 番号 | 資材名 | 準備必須 | 解説 |
| --- | --- | --- | --- |
| 5 | 処理結果を格納するクラス（図中の Result A） | 任意 | *処理結果を格納するクラスを作成する* |

## 送信するデータを格納するためのクラスを作成する

最初に、送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。

```
package jp.co.intra_mart.module_name.function_name.propagation;

import java.net.URL;

public class SenderModuleData {

    /** リソースID */
    private String resourceId;

    /** URL */
    private URL url;

    public String getResourceId() {
        return resourceId;
    }

    public URL getUrl() {
        return url;
    }

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

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

}
```

送信側から送ることができるデータを格納するクラスは1つだけです。
そのため、同時に送信したいデータのクラスが複数ある場合、1つのクラスにまとめてください。

注意

「独自モデル」のパッケージ名を含むクラス名（完全修飾子）が、データを伝搬する上でのキー情報の1つになります。
そのため、**後から容易に変更できませんので、慎重に命名するようにしてください。**
例えば、パッケージ名はモジュールや機能を示す名称、クラス名には送信するデータの内容を示す名称を指定してください。

## 送受信モデル（Generic）を作成する

次に、送信するデータを直列化するための「送受信モデル（Generic）」を作成します。
ただし、以下のいずれかに該当する場合は、このクラスを新規に作成する必要はありません。

- 「独自モデル」が「送受信モデル（Generic）」の制約を満たすクラスの場合、「独自モデル」を流用できます。

「送受信モデル（Generic）」の制約についての詳細は、「IM-Propagation 仕様書」を参照してください。

- 「独自モデル」に格納されているデータが IM-Propagation 内に定義されている「送受信モデル（Generic）」のいずれかのクラスに格納できる場合、定義済みのクラスを流用できます。
  定義されている「送受信モデル（Generic）」は、「APIドキュメント - jp.co.intra_mart.foundation.propagation.model.generic」を参照してください。

新しい「送受信モデル（Generic）」を用意する場合は、AbstractGeneric クラスを継承して新しいクラスを作成してください。
「独自モデル」で定義されているプロパティのうち送信したいデータが全て格納でき、かつ、「送受信モデル（Generic）」の制約を満たすクラスを作成してください。

```
package jp.co.intra_mart.module_name.function_name.propagation.generic;

import jp.co.intra_mart.foundation.propagation.model.generic.AbstractGeneric;

public class SampleGenericData extends AbstractGeneric {

    /** バージョン番号（新しく採番してください） */
    private static final long serialVersionUID = 1234567890123456789L;

    /** リソースID */
    private String resourceId;

    /** URL */
    private String url;

    public String getResourceId() {
        return resourceId;
    }

    public String getUrl() {
        return url;
    }

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

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

}
```

注意

AbstractGeneric クラスは Serializable インタフェースを実装しているため、serialVersionUID が必要です。
上記の値をそのまま使用せず、新しく採番してください。

## 送信側のデータ変換クラス（Encoder）を作成する

次に、「独自モデル」を「送受信モデル（Generic）」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」を作成する際は、AbstractEncoder クラスを継承してください。
AbstractEncoder クラスの総称型は、左から「独自モデル」、「送受信モデル（Generic）」の順にクラスタイプを指定

してください。

AbstractEncoder クラスを継承することで、encode、getGenericDataClass メソッドの実装が必須となります。

```
package jp.co.intra_mart.module_name.function_name.propagation.encoder;

import jp.co.intra_mart.foundation.propagation.exception.ConvertException;
import jp.co.intra_mart.foundation.propagation.sender.AbstractEncoder;

public class SampleEncoder extends AbstractEncoder<SenderModuleData, SampleGenericData> {

    /**
     * 「独自モデル」から「送受信モデル」に変換するメソッド
     * @param data 独自モデル
     * @return 送受信モデル
     * @throws 変換に失敗した場合
     */
    @Override
    public SampleGenericData encode(final SenderModuleData data) throws ConvertException {
        // 入力チェック
        if (data.getResourceId() == null || data.getUrl() == null) {
            throw new ConvertException("Required parameters are null.");
        }

        // 独自モデルから送受信モデルに変換して返却
        final SampleGenericData generic = new SampleGenericData();
        generic.setResourceId(data.getResourceId());
        generic.setUrl(data.getUrl().toString());
        return generic;
    }

    /**
     * 「送受信モデル」のクラスを返却するメソッド
     * @return 送受信モデルのクラス
     */
    @Override
    public Class<SampleGenericData> getGenericDataClass() {
        return SampleGenericData.class;
    }

}
```

実装が必要なメソッドとその説明は、以下の通りです。

- encode メソッド

  引数から渡された「独自モデル」を「送受信モデル（Generic）」に入れ替えて返却してください。
  「独自モデル」と「送受信モデル（Generic）」がまったく同じクラスの場合は、そのまま引数の値を返却することができます。

  データの入れ替えができない状況のときや、必要なデータが格納されていなかった場合は、ConvertException、または、ConvertException を継承した例外クラスをスローしてください。

- getGenericDataClass メソッド

  「送受信モデル（Generic）」のクラスを返却してください。

## マッピング設定を作成する

次に、ここまで作成した「独自モデル」「送受信モデル（Generic）」「データ変換クラス」を紐付けるためのマッピン

グ設定を作成します。
ファイル名の最初にはモジュールIDを含めるなど、他のモジュールが提供している設定ファイルと衝突しないようにしてください。

マッピング設定は、以下のような形式で記述します。
「設定ファイルリファレンス - IM-Propagation 送信側設定」も合わせて参照してください。

| パス | WEB-INF/conf/propagation-senders-config/{任意のファイル名}.xml |
|---|---|

```
<?xml version="1.0" encoding="UTF-8"?>
<propagation-senders-config xmlns="http://www.intra-mart.jp/propagation/senders-config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.intra-mart.jp/propagation/senders-config propagation-senders-config.xsd">
  <sender source="jp.co.intra_mart.module_name.function_name.propagation.SenderModuleData"
operationType="DATA_CREATED">
    <encoder class="jp.co.intra_mart.module_name.function_name.propagation.encoder.SampleEncoder" />
  </sender>
</propagation-senders-config>
```

注意

ファイル名は、他のモジュールが提供しているものと重複しないようにするために、モジュールIDを接頭子にするなどの対策を行ってください。

例） sender_module-sample_data.xml

設定が必要なタグとその説明は、以下の通りです。

- sender タグ

  source 属性には、「独自モデル」のパッケージ名を含むクラス名（完全修飾子）を指定します。

  operationType 属性には、データ変換対象の「データの操作種別」を指定します。
  指定可能な値の候補が IM-Propagation でいくつか用意されていますので、基本的には候補の中から使用してください。
  候補の一覧は「IM-Propagation 仕様書」、および、「APIドキュメント - OperationType」を参照してください。

  独自に「データの操作種別」を定義する場合は、必ずモジュールIDを接頭子とした文字列にしてください。

  例） "sender_module.DATA_UPLOADED"

- encoder タグ

  class 属性には、「データ変換クラス」のパッケージ名を含むクラス名（完全修飾子）を指定します。

コラム

「データ変換クラス」に渡すことができる独自のパラメータ文字列を指定することができます。指定は任意です。
詳細は、「*「データ変換クラス」の初期パラメータを設定する*」を参照してください。

## 処理結果を格納するクラスを作成する

最後に、受信側が処理した結果を受け取るための「処理結果を格納するクラス」を作成します。

「処理結果を格納するクラス」の指定は必須ですが、特に処理結果を受け取らない場合は、IM-Propagation で用意されている jp.co.intra_mart.foundation.propagation.model.EmptyObject クラスで代用できます。

新しい「処理結果を格納するクラス」を用意する場合は、「処理結果を格納するクラス」の制約を満たすクラスを作成してください。
「処理結果を格納するクラス」の制約についての詳細は、「IM-Propagation 仕様書 」を参照してください。

```
package jp.co.intra_mart.module_name.function_name.propagation.result;

import java.io.Serializable;

public class ProcResult implements Serializable {

    /** バージョン番号（新しく採番してください） */
    private static final long serialVersionUID = 1234567890123456789L;

    /** 成功フラグ*/
    private boolean success;

    public boolean isSuccess() {
        return success;
    }

    public void setSuccess(final boolean success) {
        this.success = success;
    }

}
```

注意

「処理結果を格納するクラス」は Serializable インタフェースを実装する必要があるため、serialVersionUID が必要です。
上記の値をそのまま使用せず、新しく採番してください。

## データを送信する

データを送信するためには、IM-Propagation マネージャのインスタンスを取得します。

```
final PropagationManager manager = PropagationManagerFactory.getInstance().getPropagationManager();
```

PropagationManager についての詳細は、「APIドキュメント - PropagationManager 」を参照してください。

IM-Propagation は独自のセッション管理を行っており、データの送信が成功・失敗した場合は正しくセッションを終了させる必要があります。
セッション管理についての詳細は、「IM-Propagation 仕様書 」を参照してください。
操作手順は以下の通りです。

1. begin メソッドを呼び出すことで、IM-Propagation のセッションが開始されます。
2. send メソッドを呼び出すことで、データが送信されます。送信したいデータが複数ある場合は send メソッドをその都度呼び出します。
3. データ送信後に decide メソッドを呼び出すことで、セッションを確定してコミット処理が行われます。
4. 1 〜 3 の処理中に何らかの例外が発生した場合は、例外処理を行います。

5. 最終処理として abort メソッドを呼び出すことで、セッションが確実に終了されます。
   このとき decide メソッドがまだ呼び出されていない場合は、セッションが中断してロールバック処理が行われます。

```
final PropagationManager manager =
PropagationManagerFactory.getInstance().getPropagationManager();
try {
    // セッションを開始
    manager.begin();

    // データを送信
    manager.send(OperationType.DATA_CREATED, SenderModuleData.class, data, EmptyObject.class);

    // セッションを確定
    manager.decide();
} catch (final BeginException e) {
    // manager.begin() に失敗した場合

} catch (final SendException e) {
    // manager.send() に失敗した場合

} catch (final DecideException e) {
    // manager.decide() に失敗した場合

} catch (final Exception e) {
    // その他例外が発生した場合

} finally {
    // セッションを中断。 manager.decide() が成功した場合は何もしない
    manager.abort();
}
```

特に例外処理を行わない場合は、クロージャ対応のメソッドを利用すると、実装を簡略化できます。
詳細は、「*クロージャ対応のメソッドを使用してデータを送信する*」を参照してください。

また、自身のモジュール内へデータを伝搬する目的で使用するなどの理由でセッション管理が不要な場合は、セッション関係の実装を省略できます。
詳細は、「*セッション管理を使用せずにデータを送信する*」を参照してください。

受信側からの処理結果を取得する方法についての詳細は、「*受信側から渡された処理結果を取得する*」を参照してください。

## 受信側へ情報を公開する

- 「独自モデル」と「データの操作種別」を公開します。

  送信したデータを受信側が取得できるようにするため、送信する「独自モデル」の完全修飾子と「データの操作種別」の組み合わせ（マッピング設定）を公開してください。

- 「送受信モデル（Generic）」と「処理結果を格納するクラス」を公開します。

  送信したデータを受信側が取得できるようにするため、「送受信モデル（Generic）」の仕様を公開してください。
  また、処理結果を受信側から受け取る場合は、受信側が処理結果を返却できるようにするため、「処理結果を格納するクラス」の仕様を公開してください。

  将来、送受信するデータの内容を変更するためにこれら2つのクラスの仕様を変更する場合は、既存のパラメータを削除したり、設定する値を変えたりすることは、できる限り避けてください。

後からパラメータを追加することは、問題ありません。

公開例：

- 「独自モデル」の完全修飾子
  jp.co.intra_mart.module_name.function_name.propagation.SenderModuleData
- データ受信のための送受信モデル
  jp.co.intra_mart.module_name.function_name.propagation.generic.SampleGenericData
- 送信側に返却するための処理結果モデル
  jp.co.intra_mart.module_name.function_name.propagation.result.ProcResult

| パラメータ | 設定値 |
|---|---|
| success | リソースIDとURLのマッピングの登録を承認する場合は true、否認する場合は false。 |

- データの操作種別とデータ送信のタイミング

| データの操作種別 | データ送信のタイミング |
|---|---|
| DATA_CREATED | リソースIDとURLのマッピングが登録された。 |

# 追加情報

## 「データ変換クラス」の初期パラメータを設定する

「IM-Propagation 送信側設定」の encoder タグ内に params、param タグを記述することで、「データ変換クラス」に渡すことができる独自のパラメータ文字列を指定することができます。
この設定により、同一の「データ変換クラス」で動的に変化させたいロジック・設定値がある場合などに、個別にクラスを分ける必要がなくなり、汎用性を向上させることができます。

```
<?xml version="1.0" encoding="UTF-8"?>
<propagation-senders-config xmlns="http://www.intra-mart.jp/propagation/senders-config"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.intra-mart.jp/propagation/senders-config propagation-senders-config.xsd">
  <sender source="jp.co.intra_mart.module_name.function_name.propagation.SenderModuleData"
operationType="DATA_CREATED">
    <encoder class="jp.co.intra_mart.module_name.function_name.propagation.encoder.SampleEncoder">
      <params>
        <param key="no-resource-id">false</param>
        <param key="no-url">true</param>
      </params>
    </encoder>
  </sender>
</propagation-senders-config>
```

設定が必要なタグとその説明は、以下の通りです。

- params タグ

  param タグの親タグです。
  このタグ内に複数の param タグを記述することで、複数のパラメータ値を引き渡すことができます。

- param タグ

  key 属性にパラメータ文字列を取得するためのキー、タグ内にパラメータ文字列を記述してください。

キーは重複可能で、取得時に同一のキーで指定された全てのパラメータ文字列を取得できます。

キーやパラメータ文字列を取得する場合は、「データ変換クラス」内で super#getParamValue()、super#getParamValues()、super#getParamKeys() メソッドを使用してください。
各メソッドについての詳細は、「APIドキュメント - AbstractEncoder 」を参照してください。

例えば、「データ変換クラス」上では以下のようにパラメータ値を取得して、処理を分岐できます。

```
package jp.co.intra_mart.module_name.function_name.propagation.encoder;

import jp.co.intra_mart.foundation.propagation.exception.ConvertException;
import jp.co.intra_mart.foundation.propagation.sender.AbstractEncoder;

public class SampleEncoder extends AbstractEncoder<SenderModuleData, SampleGenericData> {

    /**
     * 「独自モデル」から「送受信モデル」に変換するメソッド
     * @param data 独自モデル
     * @return 送受信モデル
     * @throws 変換に失敗した場合
     */
    @Override
    public SampleGenericData encode(final SenderModuleData data) throws ConvertException {
        // 入力チェック
        if (data.getResourceId() == null || data.getUrl() == null) {
            throw new ConvertException("Required parameters are null.");
        }

        // 独自モデルから送受信モデルに変換
        // 属性値によってパラメータに設定するかどうかを決める
        final SampleGenericData generic = new SampleGenericData();
        if (!"true".equals(getParamValue("no-resource-id"))) {
            generic.setResourceId(data.getResourceId());
        }
        if (!"true".equals(getParamValue("no-url"))) {
            generic.setUrl(data.getUrl().toString());
        }

        // 送受信モデルを返却
        return generic;
    }

    /**
     * 「送受信モデル」のクラスを返却するメソッド
     * @return 送受信モデルのクラス
     */
    @Override
    public Class<SampleGenericData> getGenericDataClass() {
        return SampleGenericData.class;
    }

}
```

## クロージャ対応のメソッドを使用してデータを送信する

クロージャ対応の execute メソッドを使用した場合は、セッション管理系のメソッド（send 以外）を呼び出さずに、セッション管理を自動化することができます。
IM-Propagation のセッション管理、および、データベースのトランザクション制御は、IM-Propagation 側が行いま

す。
特に細かな例外処理が求められない場合、この方法が最も簡単です。

Callable#call() メソッドで任意のオブジェクトのインスタンスが返却された場合は、decide メソッドを呼び出して自動的にセッションを確定します。
null が返却された、または、例外が発生した場合は、abort メソッドを呼び出して自動的にセッションを中断します。
Callable#call() メソッドで返却された値が、このメソッドの戻り値としてそのまま返却されます。

```
final PropagationManager manager = PropagationManagerFactory.getInstance().getPropagationManager();
try {
    // データを送信
    manager.execute(new Callable<SendResult<EmptyObject>>() {
        @Override
        public SendResult<EmptyObject> call() throws Exception {
            return manager.send(OperationType.DATA_CREATED, SenderModuleData.class, data,
EmptyObject.class);
        }
    });
} catch (final PropagationManagerException e) {
    // |common_im_propagation| で失敗した場合

} catch (final Exception e) {
    // その他例外が発生した場合

}
```

注意

Callable#call() メソッドの戻り値に null を返却した場合は、セッションが中断されますので注意してください。
特に返却するものがない場合でも、何らかのオブジェクトのインスタンス（例えば、SendResult）を返却してください。

## セッション管理を使用せずにデータを送信する

IM-Propagation のセッション管理が不要で単純にデータを伝播したい場合は、begin、decide、abort メソッドの呼び出しを省略することができます。
この場合、リソースを解放するために最終処理として close メソッドを呼び出してください。

```
final PropagationManager manager = PropagationManagerFactory.getInstance().getPropagationManager();
try {
    // データを送信
    manager.send(OperationType.DATA_CREATED, SenderModuleData.class, data, EmptyObject.class);
} catch (final SendException e) {
    // manager.send() に失敗した場合

} catch (final Exception e) {
    // その他例外が発生した場合

} finally {
    // 処理を終了
    manager.close();
}
```

注意

この実装が有効なケースは、以下のいずれかが保証されている場合に限られますので注意してください。

- データの受信側がセッションの開始（データベースのトランザクション、または、独自のセッション管理）を要求していない場合。
- 自分のモジュール内へデータを伝搬する目的で使用する場合（他のモジュールへデータを伝搬しない場合）。
- この実装の外側で IM-Propagation のセッション管理が行われている場合。

## 受信側から渡された処理結果を取得する

受信側から渡された処理結果を取得する場合の操作手順は、以下の通りです。

1. `send` メソッドの第4引数に、処理結果を格納するクラスを指定します。
2. `send` メソッドの戻り値（`SendResult` クラスのインスタンス）を取得します。
3. `SendResult#getResponses()` メソッドから、全ての受信側が返却した処理結果を取得します。

```
SendResult<ProcResult> result = manager.send(OperationType.DATA_CREATED,
SenderModuleData.class, data, ProcResult.class);
for (ProcResult response : result.getResponses()) {
    System.out.print(response.isSuccess());
}
```

受信側が処理結果を返さなかった場合、または、明示的に `null` を返した場合、`SendResult#getResponses()` メソッドから取得された一覧には含まれません（一覧内に `null` は含まれません）。

# データを受け取る側の実装

項目


- 用意する資材
- 送受信モデル（Generic）を作成する
- 受信するデータを格納するためのクラスを作成する
- 受信側のデータ変換クラス（Decoder）を作成する
- 処理結果を格納するクラスを作成する
- 受信側のデータ処理クラス（Procedure）を作成する
- マッピング設定を作成する
- データを受信する
- 追加情報
  - 「データ変換クラス」「データ処理クラス」の初期パラメータを設定する
  - データベース以外の操作を行う「データ処理クラス」の実装


## 用意する資材

受信側で用意する資材は、以下の通りです。

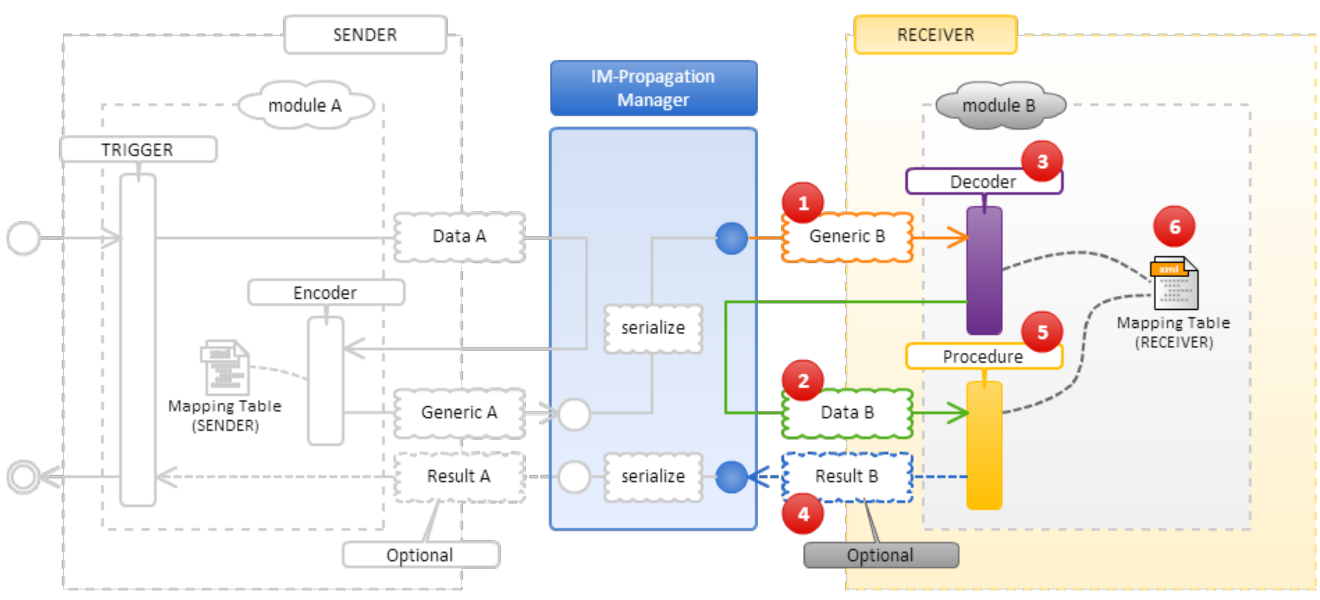


図 受信側の処理

表 用意する資材一覧

| 番号 | 資材名 | 準備必須 | 解説 |
| --- | --- | --- | --- |
| 1 | 送受信モデル（Generic）（図中の Generic B） | 必須 | *送受信モデル（Generic）を作成する* |
| 2 | 受信するデータを格納するためのクラス（独自モデル）（図中の Data B） | 必須 | *受信するデータを格納するためのクラスを作成する* |
| 3 | データ変換クラス（図中の Decoder） | 必須 | *受信側のデータ変換クラス（Decoder）を作成する* |
| 4 | 処理結果を格納するクラス（図中の Result B） | 任意 | *処理結果を格納するクラスを作成する* |

| 番号 | 資材名 | 準備必須 | 解説 |
|---|---|---|---|
| 5 | データ処理クラス（図中の Procedure） | 必須 | *受信側のデータ処理クラス（Procedure）を作成する* |
| 6 | マッピング設定（図中の Mapping Table） | 必須 | *マッピング設定を作成する* |

また、受信したいデータが送信側から送信されるタイミングを示す以下のキー情報を、送信側のマッピング設定などから入手してください。
下記の例では、「*データを送る側の実装*」で作成したデータを受信するための元となる資材を記載しています。

1. 送信する「独自モデル」の完全修飾子

   例） jp.co.intra_mart.module_name.function_name.propagation.SenderModuleData

2. データの操作種別

   例） DATA_CREATED

3. 送信側が送る「送受信モデル（Generic）」のパラメータ名と型情報

   例） jp.co.intra_mart.module_name.function_name.propagation.generic.SampleGenericData

   | パラメータ名 | 型 |
   |---|---|
   | resourceId | String |
   | url | String |

4. 送信側が受け取る「処理結果を格納するクラス」のパラメータ名と型情報
   送信側が処理結果を要求していない場合は不要です。

   例） jp.co.intra_mart.module_name.function_name.propagation.result.ProcResult

   | パラメータ名 | 型 |
   |---|---|
   | success | boolean |

## 送受信モデル（Generic）を作成する

最初に、直列化したデータを受信するための「送受信モデル（Generic）」を作成します。
ただし、以下に該当する場合は、このクラスを新規に作成する必要はありません。

- 送信されるデータが格納される、データの送信側が定義した「送受信モデル（Generic）」が IM-Propagation 内に定義されている「送受信モデル（Generic）」のいずれかのクラスで復元可能である場合、定義済みのクラスを流用できます。
  定義されている「送受信モデル（Generic）」は、「API ドキュメント - jp.co.intra_mart.foundation.propagation.model.generic」を参照してください。
  復元可能かどうかを判定する場合は、「*復元可能なクラスとは*」を参照してください。

新しい「送受信モデル（Generic）」を用意する場合は、AbstractGeneric クラスを継承して新しいクラスを作成してください。
送信側が公開している「送受信モデル（Generic）」から受信したいデータが全て格納でき、かつ、「送受信モデル（Generic）」の制約を満たすクラスを作成してください。
「送受信モデル（Generic）」の制約についての詳細は、「IM-Propagation 仕様書」を参照してください。

```java
package jp.co.intra_mart.module_name.function_name.propagation.generic;

import jp.co.intra_mart.foundation.propagation.model.generic.AbstractGeneric;

public class SampleGenericData extends AbstractGeneric {

    /** バージョン番号（新しく採番してください） */
    private static final long serialVersionUID = 1234567890123456789L;

    /** リソースID */
    private String resourceId;

    /** URL */
    private String url;

    public String getResourceId() {
        return resourceId;
    }

    public String getUrl() {
        return url;
    }

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

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

}
```

注意

AbstractGeneric クラスは Serializable インタフェースを実装しているため、serialVersionUID が必要です。
上記の値をそのまま使用せず、新しく採番してください。

## 受信するデータを格納するためのクラスを作成する

次に、後で作成する「受信側のデータ変換クラス（Decoder）」を通して、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。
例えば、「送受信モデル（Generic）」において String 型で定義されたプロパティを、「独自モデル」においては URL 型として扱いたい場合、String から URL に変換処理を行うために変換後のデータを格納するためのクラスが必要になります。

```
package jp.co.intra_mart.module_name.function_name.propagation;

import java.net.URL;

public class ReceiverModuleData {

    /** リソースID */
    private String resourceId;

    /** URL */
    private URL url;

    public String getResourceId() {
        return resourceId;
    }

    public URL getUrl() {
        return url;
    }

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

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

}
```

## 受信側のデータ変換クラス（Decoder）を作成する

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」を作成する際は、`AbstractDecoder` クラスを継承してください。
`AbstractDecoder` クラスの総称型は、左から「送受信モデル（Generic）」、「独自モデル」の順にクラスタイプを指定してください。

`AbstractDecoder` クラスを継承することで、`decode`、`getGenericDataClass` メソッドの実装が必須となります。

```java
package jp.co.intra_mart.module_name.function_name.propagation.decoder;

import java.net.MalformedURLException;
import java.net.URL;

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

public class SampleDecoder extends AbstractDecoder<SampleGenericData, ReceiverModuleData> {

    /**
     * 「送受信モデル」から「独自モデル」に変換するメソッド
     * @param generic 送受信モデル
     * @return 独自モデル
     * @throws 変換に失敗した場合
     */
    @Override
    public ReceiverModuleData decode(final SampleGenericData generic) throws ConvertException {
        try {
            // 送受信モデルから独自モデルに変換
            final ReceiverModuleData data = new ReceiverModuleData();
            if (generic.getResourceId() != null) {
                data.setResourceId(generic.getResourceId());
            }
            if (generic.getUrl() != null) {
                data.setUrl(new URL(generic.getUrl()));
            }

            // 独自モデルを返却
            return data;
        } catch (final MalformedURLException e) {
            throw new ConvertException(e);
        }
    }

    /**
     * 「送受信モデル」のクラスを返却するメソッド
     * @return 送受信モデルのクラス
     */
    @Override
    public Class<SampleGenericData> getGenericDataClass() {
        return SampleGenericData.class;
    }

}
```

実装が必要なメソッドとその説明は、以下の通りです。

- `decode` メソッド

  引数から渡された「送受信モデル（Generic）」を「独自モデル」に入れ替えて返却してください。
  「送受信モデル（Generic）」と「独自モデル」がまったく同じクラスの場合は、そのまま引数の値を返却することができます。

  データの入れ替えができない状況のときや、必要なデータが格納されていなかった場合は、`ConvertException` 、または、`ConvertException` を継承した例外クラスをスローしてください。

- `getGenericDataClass` メソッド

  「送受信モデル（Generic）」のクラスを返却してください。

## 処理結果を格納するクラスを作成する

次に、受信側が処理した結果を送り返すための「処理結果を格納するクラス」を作成します。

特に処理結果を返却しない場合はクラスを用意する必要はなく、IM-Propagation で用意されている jp.co.intra_mart.foundation.propagation.model.EmptyObject クラスで代用できます。

新しい「処理結果を格納するクラス」を用意する場合は、送信側が公開している「処理結果を格納するクラス」から復元可能なクラスを作成してください。
復元可能かどうかを判定する場合は、「*復元可能なクラスとは*」を参照してください。

```
package jp.co.intra_mart.module_name.function_name.propagation.result;

import java.io.Serializable;

public class ProcResult implements Serializable {

    /** バージョン番号（新しく採番してください） */
    private static final long serialVersionUID = 1234567890123456789L;

    /** 成功フラグ */
    private boolean success;

    public boolean isSuccess() {
        return success;
    }

    public void setSuccess(final boolean success) {
        this.success = success;
    }

}
```

注意

「処理結果を格納するクラス」は Serializable インタフェースを実装する必要があるため、serialVersionUID が必要です。
上記の値をそのまま使用せず、新しく採番してください。

## 受信側のデータ処理クラス（Procedure）を作成する

次に、「送受信モデル（Generic）」または「独自モデル」を受け取りデータ処理を行うクラス（以下、「データ処理クラス」）を用意します。

IM-Propagation は独自のセッション管理を行っており、データベースのトランザクション管理は IM-Propagation が自動的に行います。
セッション管理についての詳細は、「IM-Propagation 仕様書」を参照してください。

「データ処理クラス」を作成する際は、AbstractProcedure クラスを継承してください。
AbstractProcedure クラスの総称型は、左から「独自モデル」、「処理結果を格納するクラス」の順にクラスタイプを指定してください。

AbstractProcedure クラスを継承することで、onReceive メソッドの実装が必須となります。

```
package jp.co.intra_mart.module_name.function_name.propagation.procedure;

import jp.co.intra_mart.foundation.propagation.code.EventStatus;
import jp.co.intra_mart.foundation.propagation.exception.ProcedureException;
import jp.co.intra_mart.foundation.propagation.model.ReceiveParameter;
import jp.co.intra_mart.foundation.propagation.model.ReceiveResult;
import jp.co.intra_mart.foundation.propagation.receiver.AbstractProcedure;

public class SampleProcedure extends AbstractProcedure<ReceiverModuleData, ProcResult> {

    /**
     * データを受信した際の処理を行うメソッド
     * @param parameter 受信時のパラメータ情報
     * @param data 独自モデル
     * @throws ProcedureException 処理に失敗した場合
     */
    @Override
    public ReceiveResult<ProcResult> onReceive(final ReceiveParameter parameter,
            final ReceiverModuleData data) throws ProcedureException {
        // データリクエストからデータ変換クラスを通してモデルを取得
        System.out.print(data.getUrl());
        System.out.print(data.getResourceId());

        // 処理成功を返却
        final ProcResult result = new ProcResult();
        result.setSuccess(true);
        return new ReceiveResult<ProcResult>(EventStatus.SUCCEEDED, result);
    }

}
```

実装が必要なメソッドとその説明は、以下の通りです。

- onReceive メソッド

  受信したデータを処理する内容を実装してください。
  メソッドの戻り値には、ReceiveResult クラスのインスタンスを作成して返却します。
  総称型には「処理結果を格納するクラス」を指定してください。

各メソッドで返却する戻り値のインスタンスを作成する際、以下のいずれかのステータス値をコンストラクタに設定してください。

表 ステータス一覧

| 値 | 概要 | 使用例 |
|---|---|---|
| SUCCEEDED | 処理成功 | 処理が最後まで正常に終了したことを示す目的で使用します。 |
| FAILED | 処理失敗 | 処理が途中で失敗したことを示す目的で使用します。 例外発生時と同様です。 |

各メソッド内で処理に失敗し例外をスローする場合は ProcedureException、または、ProcedureException を継承した例外クラスをスローしてください。

コラム

1. SUCCEEDED、FAILED 以外のステータス値は、IM-Propagation マネージャ内部で使用します。明示的には使用しません。
2. 「処理結果を格納するクラス」を作成せず、特に処理結果を返さない場合は、最後の ReceiveResult インスタンスの返却を以下のように実装することができます。

```
return new ReceiveResult<EmptyObject>(EventStatus.SUCCEEDED);
```

注意

onReceive メソッド内でデータベース以外（ファイルやメモリなど）の更新・削除操作を行う場合は、データベースとは別のトランザクション管理を実装する必要があります。
詳細は、「*データベース以外の操作を行う「データ処理クラス」の実装*」を参照してください。

## マッピング設定を作成する

次に、ここまで作成した「送受信モデル（Generic）」「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。
ファイル名の最初にはモジュールIDを含めるなど、他のモジュールが提供している設定ファイルと衝突しないようにしてください。

マッピング設定は、以下のような形式で記述します。
「設定ファイルリファレンス - IM-Propagation 受信側設定」も合わせて参照してください。

| パス | WEB-INF/conf/propagation-receivers-config/{任意のファイル名}.xml |
|---|---|

```
<?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="jp.co.intra_mart.module_name.function_name.propagation.decoder.SampleDecoder" />
    <procedure class="jp.co.intra_mart.module_name.function_name.propagation.procedure.SampleProcedure" />
  </receiver>
</propagation-receivers-config>
```

注意

ファイル名は、他のモジュールが提供しているものと重複しないようにするために、モジュールIDを接頭子にするなどの対策を行ってください。

例） receiver_module-sample_data.xml

設定が必要なタグとその説明は、以下の通りです。

- receiver タグ

  source 属性には、**データを送信する側が指定した**「独自モデル」、または、「送受信モデル（Generic）」の

パッケージ名を含むクラス名（完全修飾子）を指定します。
**受信側のクラス名ではありませんので注意してください。**

例えば、送信側が定義した SenderModuleData クラスのデータを受信したい場合は、SenderModuleData のクラス名を指定します。キーとして使用されるだけですので、この場合でも送信側モジュールとの依存関係は必要ありません。

operationType 属性には、**データを送信する側が指定した**「データの操作種別」を指定します。

例えば、送信側が定義した SenderModuleData クラスのデータが、DATA_CREATED の「データの操作種別」で送信される場合、受信側も同じ値 DATA_CREATED を指定します。

- decoder タグ

  class 属性には、「データ変換クラス」のパッケージ名を含むクラス名（完全修飾子）を指定します。

- procedure タグ

  class 属性には、「データ処理クラス」のパッケージ名を含むクラス名（完全修飾子）を指定します。

コラム

「データ変換クラス」や「データ処理クラス」に渡すことができる独自のパラメータ文字列を指定することができます。指定は任意です。
詳細は、「 *「データ変換クラス」「データ処理クラス」の初期パラメータを設定する* 」を参照してください。

## データを受信する

各資材をアプリケーションサーバ内に配置した後、受信対象のデータが送信されるよう操作を行ってください。
正しく設定が行われていれば、「データ処理クラス」の onReceive メソッドが呼び出され、処理が行われます。
例えばサンプル（SampleProcedure）の場合、受信したクラス内に含まれる url と resourceId の内容がコンソールに出力されます。

送信操作を行っても正しく受信できない場合は、以下の点を確認してください。

- マッピング設定の記載が間違っていないか。
- 「送受信モデル（Generic）」が、送信側が定義するクラスから復元可能かどうか。
- 「データ変換クラス」が、AbstractDecoder クラスを継承しているか。
- 「データ処理クラス」が、AbstractProcedure クラスを継承しているか。

## 追加情報

### 「データ変換クラス」「データ処理クラス」の初期パラメータを設定する

「IM-Propagation 受信側設定」の decoder タグ内、および、procedure タグ内に params、param タグを記述することで、「データ変換クラス」や「データ処理クラス」に渡すことができる独自のパラメータ文字列を指定することができます。
この設定により、同一の「データ変換クラス」や「データ処理クラス」で動的に変化させたいロジック・設定値がある場合などに、個別にクラスを分ける必要がなくなり、汎用性を向上させることができます。

```
<?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="jp.co.intra_mart.module_name.function_name.propagation.decoder.SampleDecoder">
      <params>
        <param key="no-resource-id">sampleValue1</param>
        <param key="no-url">sampleValue2</param>
      </params>
    </decoder>
    <procedure class="jp.co.intra_mart.module_name.function_name.propagation.procedure.SampleProcedure">
      <params>
        <param key="update-keys">key1</param>
        <param key="update-keys">key2</param>
      </params>
    </procedure>
  </receiver>
</propagation-receivers-config>
```

設定が必要なタグとその説明は、以下の通りです。

- params タグ

  param タグの親タグです。
  このタグ内に複数の param タグを記述することで、複数のパラメータ値を引き渡すことができます。

- param タグ

  key 属性にパラメータ文字列を取得するためのキー、タグ内にパラメータ文字列を記述してください。
  キーは重複可能で、取得時に同一のキーで指定された全てのパラメータ文字列を取得できます。

  キーやパラメータ文字列を取得する場合は、「データ変換クラス」と「データ処理クラス」内で super#getParamValue()、super#getParamValues()、super#getParamKeys() メソッドを使用してください。
  各メソッドについての詳細は、「APIドキュメント - AbstractDecoder 」「APIドキュメント - AbstractSessionableProcedure 」を参照してください。

例えば、「データ変換クラス」上では以下のようにパラメータ値を取得して、処理を分岐できます。

```
package jp.co.intra_mart.module_name.function_name.propagation.decoder;

import java.net.MalformedURLException;
import java.net.URL;

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

public class SampleDecoder extends AbstractDecoder<SampleGenericData, ReceiverModuleData> {

    /**
     * 「送受信モデル」から「独自モデル」に変換するメソッド
     * @param generic 送受信モデル
     * @return 独自モデル
     * @throws 変換に失敗した場合
     */
    @Override
    public ReceiverModuleData decode(final SampleGenericData generic) throws ConvertException {
        try {
            // 送受信モデルから独自モデルに変換
            final ReceiverModuleData data = new ReceiverModuleData();
            if (generic.getResourceId() != null && !"true".equals(getParamValue("no-resource-id"))) {
                data.setResourceId(generic.getResourceId());
            }
            if (generic.getUrl() != null && !"true".equals(getParamValue("no-url"))) {
                data.setUrl(new URL(generic.getUrl()));
            }

            // 独自モデルを返却
            return data;
        } catch (final MalformedURLException e) {
            throw new ConvertException(e);
        }
    }

    /**
     * 「送受信モデル」のクラスを返却するメソッド
     * @return 送受信モデルのクラス
     */
    @Override
    public Class<SampleGenericData> getGenericDataClass() {
        return SampleGenericData.class;
    }

}
```

「データ処理クラス」内でも同様に、すべてのメソッド内で使用できます。

## データベース以外の操作を行う「データ処理クラス」の実装

データベースのトランザクション管理は IM-Propagation が自動的に行いますが、データベース以外のデータ操作を行う場合は、データの受信側が独自にトランザクション管理を行う必要があります。
例えばファイル操作の場合、自身のモジュール側の処理が成功したとしても、他のモジュールで処理が失敗してしまった場合は、それまでの操作を戻す（ロールバック）処理が必要になります。

このような独自のトランザクション処理を含む「データ処理クラス」を作成する際は、AbstractProcedure ではなく AbstractSessionableProcedure クラスを継承してください。
AbstractSessionableProcedure クラスの総称型は、左から「独自モデル」、「処理結果を格納するクラス」の順にクラスタイプを指定してください。

AbstractSessionableProcedure クラスを継承することで、onInitialize、onReceive、onPrepare、onDecide、および、onAbort メソッドの実装が必須となります。

```
package jp.co.intra_mart.module_name.function_name.propagation.procedure;

import jp.co.intra_mart.foundation.propagation.code.EventStatus;
import jp.co.intra_mart.foundation.propagation.exception.ProcedureException;
import jp.co.intra_mart.foundation.propagation.model.AbortParameter;
import jp.co.intra_mart.foundation.propagation.model.AbortResult;
import jp.co.intra_mart.foundation.propagation.model.DecideParameter;
import jp.co.intra_mart.foundation.propagation.model.DecideResult;
import jp.co.intra_mart.foundation.propagation.model.EmptyObject;
import jp.co.intra_mart.foundation.propagation.model.InitializeParameter;
import jp.co.intra_mart.foundation.propagation.model.InitializeResult;
import jp.co.intra_mart.foundation.propagation.model.PrepareParameter;
import jp.co.intra_mart.foundation.propagation.model.PrepareResult;
import jp.co.intra_mart.foundation.propagation.model.ReceiveParameter;
import jp.co.intra_mart.foundation.propagation.model.ReceiveResult;
import jp.co.intra_mart.foundation.propagation.receiver.AbstractSessionableProcedure;

public class SampleProcedure extends AbstractSessionableProcedure<ReceiverModuleData, EmptyObject>
{

    /**
     * セッションが開始した際の処理を行うメソッド
     * @param parameter 受信時のパラメータ情報
     * @throws ProcedureException 処理に失敗した場合
     */
    @Override
    public InitializeResult onInitialize(final InitializeParameter parameter) throws ProcedureException {
        // 一時データ格納領域の初期化
        return new InitializeResult(EventStatus.NOT_AFFECTED);
    }

    /**
     * データを受信した際の処理を行うメソッド
     * @param parameter 受信時のパラメータ情報
     * @param data 独自モデル
     * @throws ProcedureException 処理に失敗した場合
     */
    @Override
    public ReceiveResult<EmptyObject> onReceive(final ReceiveParameter parameter,
            final ReceiverModuleData data) throws ProcedureException {
        // onInitialize からの変更を一時データとして管理して、いつでもロールバックできるようにする
        return new ReceiveResult<EmptyObject>(EventStatus.NOT_AFFECTED);
    }

    /**
     * セッションが確定する前段階の処理を行うメソッド
     * @param parameter 受信時のパラメータ情報
     * @throws ProcedureException 処理に失敗した場合
     */
    @Override
    public PrepareResult onPrepare(PrepareParameter parameter) throws ProcedureException {
        // トランザクションが確定してデータが更新できるかどうかを判定
        // SUCCEEDED を返す場合は、次に onDecide または onAbort が呼び出されるまでロックをかけ、
        // データの衝突が起こらないよう (onDecide で必ず成功するよう) 対処する
        return new PrepareResult(EventStatus.NOT_AFFECTED);
    }

    /**
     * セッションが確定した際の処理を行うメソッド
```

```
     * @param parameter 受信時のパラメータ情報
     * @throws ProcedureException 処理に失敗した場合
     */
    @Override
    public DecideResult onDecide(final DecideParameter parameter) throws ProcedureException {
        // onPrepare が呼び出された時点での一時データを確定して本更新をかける
        return new DecideResult(EventStatus.NOT_AFFECTED);
    }

    /**
     * セッションが中断した際の処理を行うメソッド
     * @param parameter 受信時のパラメータ情報
     * @throws ProcedureException 処理に失敗した場合
     */
    @Override
    public AbortResult onAbort(final AbortParameter parameter) throws ProcedureException {
        // onInitiaize から行われた一時データへの変更を破棄
        return new AbortResult(EventStatus.NOT_AFFECTED);
    }

}
```

実装が必要なメソッドとその説明は、以下の通りです。

- onInitialize メソッド

  「データ処理クラス」の初期化処理を実装してください。
  メソッドの戻り値には、InitializeResult クラスのインスタンスを作成して返却します。

- onReceive メソッド

  受信したデータを処理する内容を実装してください。
  メソッドの戻り値には、ReceiveResult クラスのインスタンスを作成して返却します。
  総称型には「処理結果を格納するクラス」を指定してください。

- onPrepare メソッド

  「データ処理クラス」内でのデータ処理を確定する準備の処理を実装してください。
  メソッドの戻り値には、PrepareResult クラスのインスタンスを作成して返却します。

  このメソッドは、onDecide メソッドが呼び出される前に、コミットが可能かどうかの問い合わせに使用されます。

- onDecide メソッド

  「データ処理クラス」内でのデータ処理を確定する（コミット）処理を実装してください。
  メソッドの戻り値には、DecideResult クラスのインスタンスを作成して返却します。

- onAbort メソッド

  「データ処理クラス」内でのデータ処理を元に戻す（ロールバック）処理を実装してください。
  メソッドの戻り値には、AbortResult クラスのインスタンスを作成して返却します。

例えば、ファイルに受信したデータを書き込む処理を実装する場合は、以下のような処理の流れになります。

1. onInitialize メソッドで、テンポラリファイルを書き込む準備を行います（ディレクトリの作成など）。
2. onReceive メソッドで、テンポラリファイルを作成し、受信したデータを書き込みます。
3. onPrepare メソッドで、他のスレッドから本番データファイルへの変更を阻止するためのロックを取得します。
4. onDecide メソッドで、テンポラリファイルの内容を本番データファイルへ書き込みます。

その後、テンポラリファイルを破棄し、本番データファイルのロックを解除します。

5. `onAbort` メソッドで、テンポラリファイルを破棄します。
   本番データファイルのロックが行われている場合は、ロックを解除します。

# 付録

項目


- 復元可能なクラスとは
- 例外クラス
- セッションのクローズ漏れを探す


## 復元可能なクラスとは

復元可能なクラスとは、直列化されたクラス内のデータが他のクラスで復元を行った際に、直列化前の情報を取得可能であるクラスです。
データの伝搬途中でクラスのインスタンス内容を直列化するため、送信側と受信側では必ずしも同一クラスを使用する必要はありません。

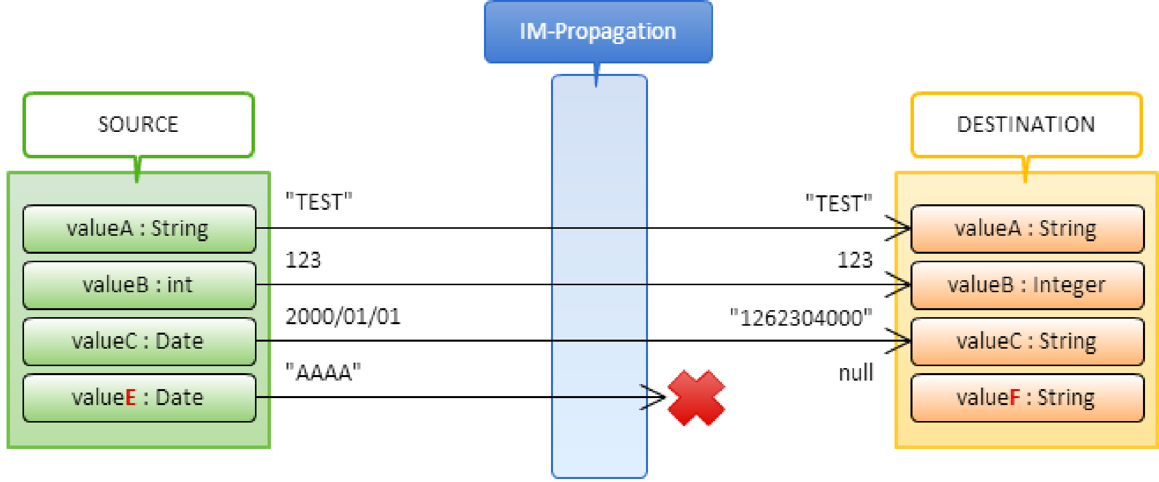


図 クラスの復元イメージ

注意

構造がまったく違うクラスの場合、直列化データを復元した際にほとんどの情報が失われる場合があります。
直列化の元となるクラスと復元に使用されるクラスの関係上で情報が失われる例は、以下の通りです。

- 同じ名前のパラメータが存在しない。
- 同じ名前のパラメータでも、型が異なる。

詳細は、「IM-Propagation 仕様書」を参照してください。

## 例外クラス

- 「送信側のデータ変換クラス（Encoder）」「受信側のデータ変換クラス（Decoder）」で独自の例外クラスをスローする場合

  ConvertException を継承した例外クラスを作成してスローしてください。
  通常は ConvertException をスローしてください。

- 「受信側のデータ処理クラス（Procedure）」で独自の例外クラスをスローする場合

  ProcedureException を継承した例外クラスを作成してスローしてください。
  通常は ProcedureException をスローしてください。

- PropagationManagerException と、PropagationManagerException を継承している例外クラスは、スローしないでください（IM-Propagation マネージャ専用のため）。

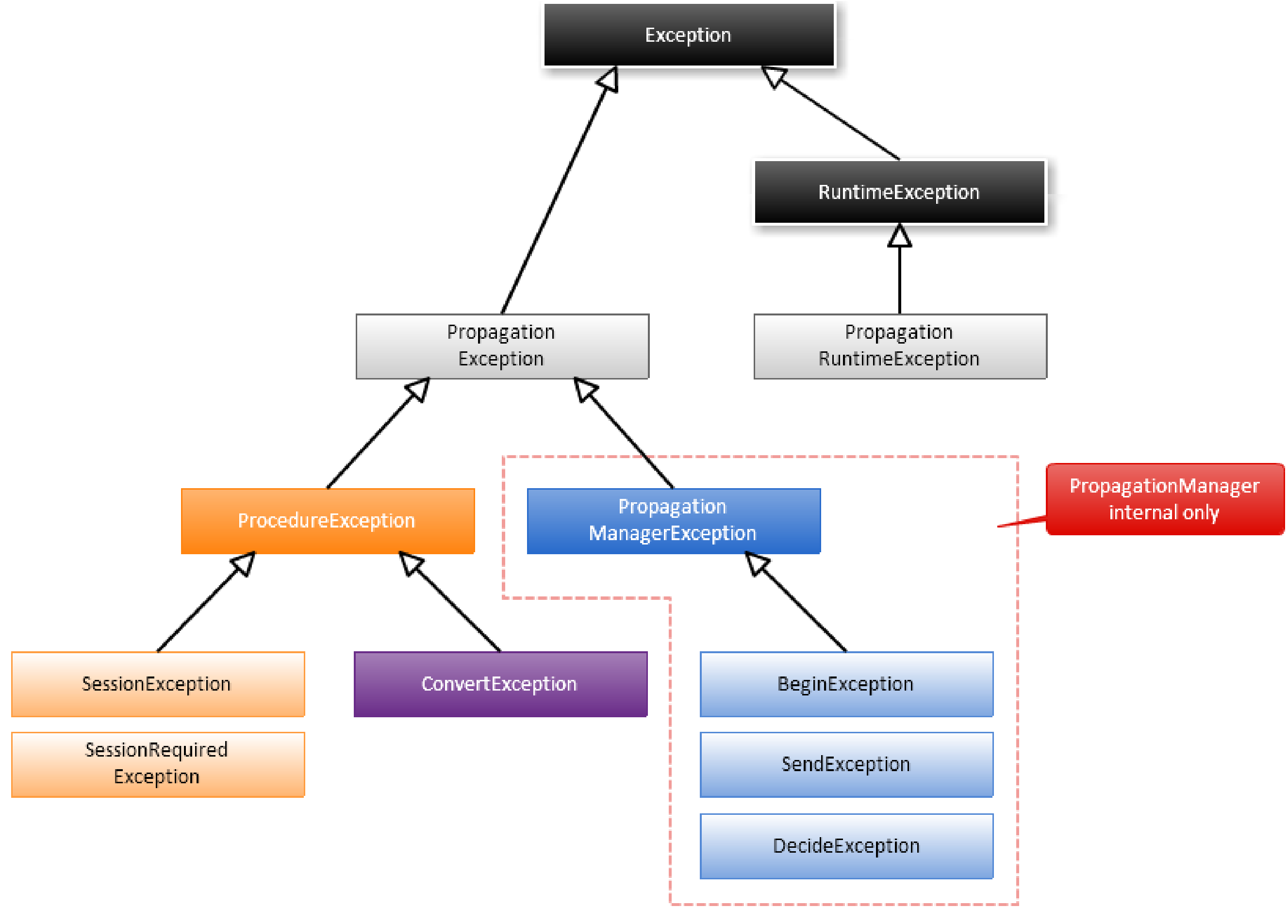


図 例外クラスの関係

## セッションのクローズ漏れを探す

IM-Propagation は内部でセッション管理を行っているため、 decide、abort、close メソッドのいずれかを呼び出して、必ずクローズ処理を行う必要があります。

クローズ処理を行わないまま IM-Propagation マネージャのインスタンスを破棄した場合は、以下の誤動作が発生する可能性があります。

- データベースのトランザクションがいつまでも終わらない。
- IM-Propagation を使用している箇所で無限ループが発生しハングアップする。

クローズ処理が行われていない箇所を探すには、以下の手順で操作してください。

1. トレースログを有効に設定します。
   ＜WEB-INF/conf/log/im_logger.xml＞ に以下を追記します。

```
<logger name="jp.co.intra_mart.system.propagation">
  <level value="trace" />
</logger>
```

2. アプリサーバを再起動します。
3. 誤動作が発生する状態を再現します。すると、以下のようなログが出力されます。

```
 1 | [DEBUG] j.c.i.s.p.PropagationManagerFactoryImpl - [] use primary. thread id=46
 2 | [DEBUG] j.c.i.s.p.PropagationManagerFactoryImpl - []   stack trace:
 3 |
jp.co.intra_mart.system.propagation.PropagationManagerFactoryImpl.writeStackTrace(PropagationManagerF

 4 |
jp.co.intra_mart.system.propagation.PropagationManagerFactoryImpl.getPropagationManager(PropagationM

 5 |     jp.co.intra_mart.module_name.function_name.propagation.Trigger.invoke(Trigger.java:43)
 6 |          :
 7 |          :
 8 | [DEBUG] j.c.i.s.p.PropagationManagerFactoryImpl - []   status map={"46":{}}
 9 | [DEBUG] j.c.i.s.p.PropagationManagerPrimary - [] called method. name=begin
10 | [DEBUG] j.c.i.s.p.PropagationManagerFactoryImpl - [] begin manager. thread id=46
11 | [DEBUG] j.c.i.s.p.PropagationManagerFactoryImpl - []   status map={"46":{}}
12 | [DEBUG] j.c.i.s.p.PropagationManagerFactoryImpl - [] primary already used, use secondary. thread
id=46
13 | [DEBUG] j.c.i.s.p.PropagationManagerFactoryImpl - []   stack trace:
14 |
jp.co.intra_mart.system.propagation.PropagationManagerFactoryImpl.writeStackTrace(PropagationManagerF

15 |
jp.co.intra_mart.system.propagation.PropagationManagerFactoryImpl.getPropagationManager(PropagationM

16 |     jp.co.intra_mart.module_name.function_name.propagation.Trigger.invoke(Trigger.java:43)
17 |          :
18 |          :
```

4. デバッグログから、以下の文字列を探します。
   上記ログの例では、12行目に存在します。

```
primary already used, use secondary.
```

5. その手前の行にある、以下の文字列を探します。
   上記ログの例では、1行目に存在します。

```
use primary.
```

6. そのすぐ後の行にスタックトレースが出力されていますので、その中から以下の文字列を探します。
   上記ログの例では、4行目に存在します。

```
jp.co.intra_mart.system.propagation.PropagationManagerFactoryImpl.getPropagationManager
```

7. そのすぐ下の行に表示されている実装箇所で、IM-Propagation マネージャのインスタンスが作成されていますが、クローズ処理が行われていない可能性があります。
   上記ログの例では、5行目の位置でクローズ処理が行われていない可能性があります。

```
jp.co.intra_mart.module_name.function_name.propagation.Trigger.invoke(Trigger.java:43)
```

   この付近の実装を重点的にご確認ください。
   この実装箇所でクローズ処理が正しく行われている場合は、4〜6の手順を別のログの場所で行ってください。