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
  - 本書の構成
- 3. 概要
  - 目的
  - 処理順序
  - 制限事項
- 4. 処理仕様
  - 全体の流れ
  - 送信側の処理
  - 受信側の処理
  - 直列化（serialize）の処理
  - マッピングの処理
- 5. セッション管理
  - セッション管理の概要
  - 送信側の対処
  - 受信側の対処
- 6. 付録
  - デッドロック管理

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。

- IM-Propagation 機能の目的
- IM-Propagation 機能の処理の流れ
- その他、IM-Propagation の仕様

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

本書の構成

- *概要*

  IM-Propagation 機能の目的について説明します。

- *処理仕様*

  IM-Propagation 機能の処理の流れについて説明します。

- *セッション管理*

  IM-Propagation 機能が独自に管理するセッション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。

概要

項目

- 目的
- 処理順序
- 制限事項

目的

IM-Propagation は、データの更新や操作の完了などを通知するための汎用的なトリガとリスナの機構を提供します。

intra-mart Accel Platform では、メニューやジョブなどのマスタデータが変更されたことを示す通知を行う際に IM-Propagation を使用しているため、主にデータ変更のリスナを用意する目的で使用します。
IM-Propagation には、通知を送信・受信するため基本的な実装が用意されており、開発者はマッピング設定といくつかの実装を追加するだけで、目的のトリガ・リスナを用意することができます。

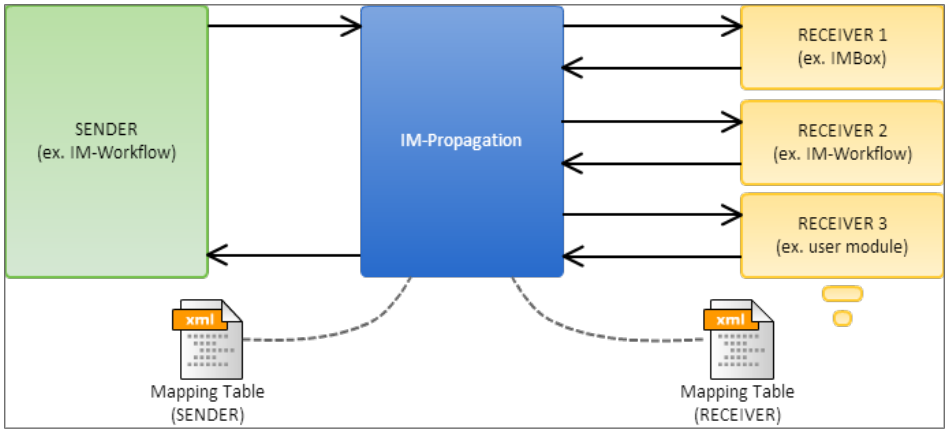

図 IM-Propagation の通知イメージ

送信側（図中の SENDER）が通知内容を IM-Propagation に送信すると、IM-Propagation はマッピング設定（図中の Mapping Table）によって受信側（図中の RECEIVER 1～3）を決定し、通知内容を伝搬します。
したがって、データや通知を受信したい場合は、マッピング設定を追加します。
受信側が複数存在する場合は1つずつ通知内容を送り、全ての受信側に通知が伝搬された後、送信側に戻ります。

データや通知を送信したい場合も同様に、マッピング設定を追加することで実現できます。

処理順序

受信側が複数存在する場合、見つかった順から1つずつ処理されます（順序は不定です）。

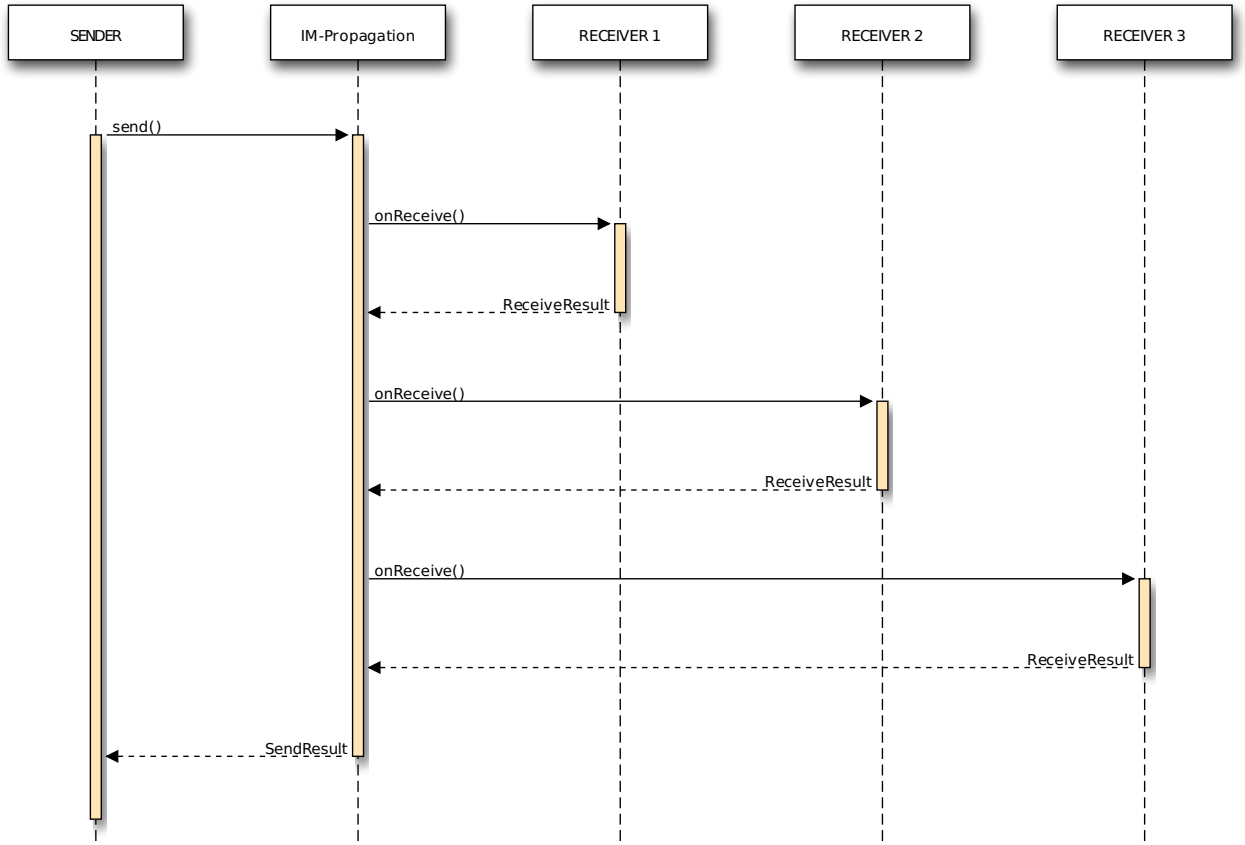

図 処理シーケンス

複数のスレッドから、同一の受信側へデータが伝搬される場合、1スレッドずつ処理されます。
これにより、受信側の処理はスレッドセーフが保証されます。

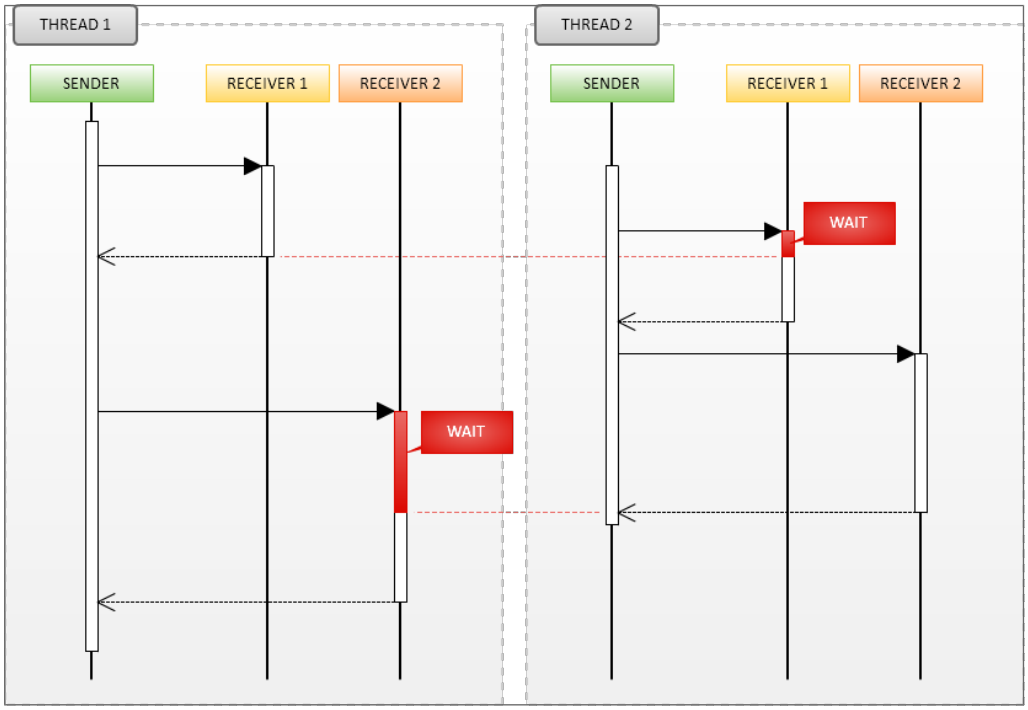

図 マルチスレッドでの動作

制限事項

- IM-Propagation を使用したデータの伝搬は、データを送信したサーバ内で処理が実行されます。
  アプリケーションサーバが複数台構成の場合、他サーバへのデータの伝搬はできません。
- IM-Propagation の処理全体でロックする機能は提供されていません。
  ロックを必要とする場合は、別途アプリケーションロックを使用します。

処理仕様

項目

- 全体の流れ
- 送信側の処理
  - 処理の流れ
  - 送信処理の手順
  - 登録済みの「データの操作種別」一覧
- 受信側の処理
  - 処理の流れ
  - 受信処理の手順
  - 返却可能な結果ステータス
- 直列化（serialize）の処理
  - 直列化の概要
  - 変換ルール
- マッピングの処理
  - クラスの解決
  - データ変換クラスとデータ処理クラスの関係

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。

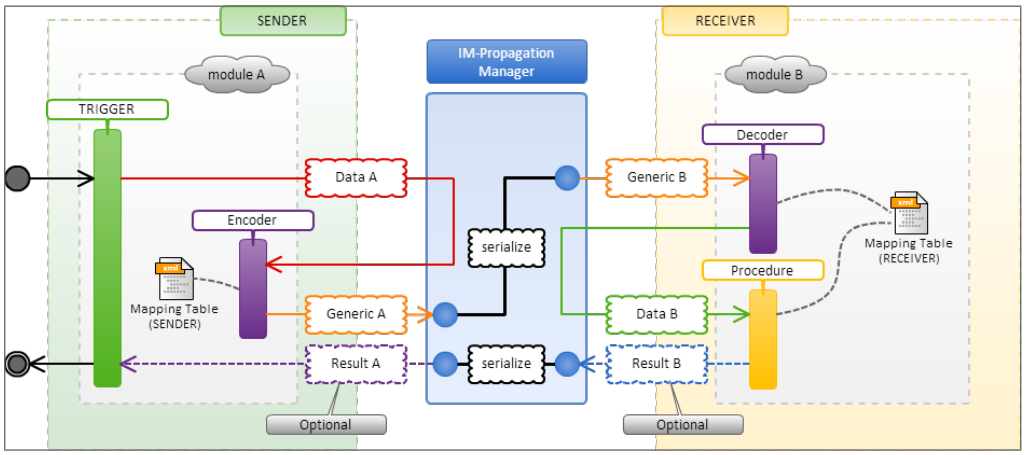

図 データの流れ

処理全体では、おおまかに「送信側」、「受信側」、および、それらを接続する「IM-Propagation マネージャ」の3つに分けられます。

- 送信側（図中の SENDER）

  IM-Propagation を使用して、通知やデータを送ります。
  送信側を構成する資材は、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）

  詳細は「*送信側の処理* 」を参照してください。

- 受信側（図中の RECEIVER）

  IM-Propagation を使用して送られた通知やデータを、送信側から受け取ります。
  受信側を構成する資材は、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. 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）
5. 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）
6. 処理結果を格納するクラス（図中の Result B）

詳細は「*受信側の処理*」を参照してください。

- IM-Propagation マネージャ（図中の IM-Propagation Manager）

  IM-Propagation マネージャでは、送信側から受け取ったデータ（図中の Generic A）を、受信側が定義するデータを格納するクラス（図中の Generic B）に変換します。
  送信側から受信側へデータを送信する際、内部でデータの直列化（図中の serialize）を行います。

  また、受信側から処理結果が返却された場合は、同様に受信側の処理結果（図中の Result B）を送信側の処理結果（図中の Result A）に変換します。

  詳細は「*直列化（serialize）の処理*」を参照してください。

送信側の処理

処理の流れ

送信側のデータ処理の流れは、以下の通りです。

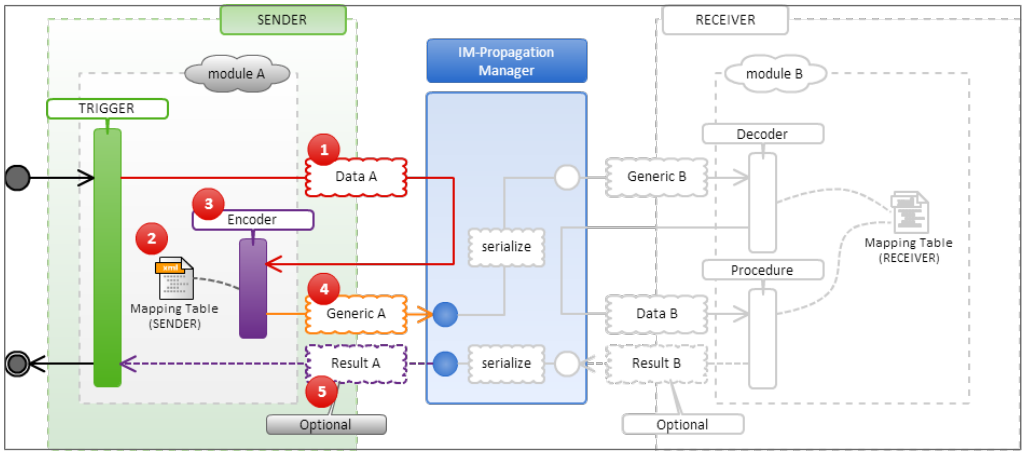

図 送信側の処理

1. あるデータを送信しようとするモジュール A（図中の module A）がデータを格納するためのクラス （以下、独自モデル。図中の Data A）を用意して、トリガー（図中の TRIGGER）から IM-Propagation マネージャに対して対象のデータをクラス内に格納して渡します。
   トリガーは IM-Propagation マネージャを使用してデータ送信処理を実装しているクラスです。
   「独自モデル」はモジュール A 側が自由に作成することができるクラスです。
2. データを受け取った IM-Propagation マネージャは、送られてきたデータを処理できる「送信側のデータ変換クラス（Encoder）」（図中の Encoder）をマッピング設定（図中の Mapping Table）から決定します。
   マッピング設定からの決定方法については、「*マッピングの処理*」を参照してください。
3. IM-Propagation マネージャが「送信側のデータ変換クラス（Encoder）」（図中の Encoder）のインスタンスを作成し、「独自モデル」のインスタンスを渡します。
4. 「送信側のデータ変換クラス（Encoder）」では、「独自モデル」のインスタンス内の情報を、IM-Propagation 内でデータをやりとりするための「送受信モデル（Generic）」（図中の Generic A）に変換します。
   「送受信モデル（Generic）」は IM-Propagation マネージャ内で処理され、受信側（図中の RECEIVER）へと渡されます。
5. 受信側から何らかの処理結果を受け取りたい場合は、処理結果を格納するクラス（図中の Result A）を用意して、IM-Propagation マネージャにデータを送る際に、処理結果を格納するクラスを指定します。
   処理結果を特に受け取らない場合は、不要です。

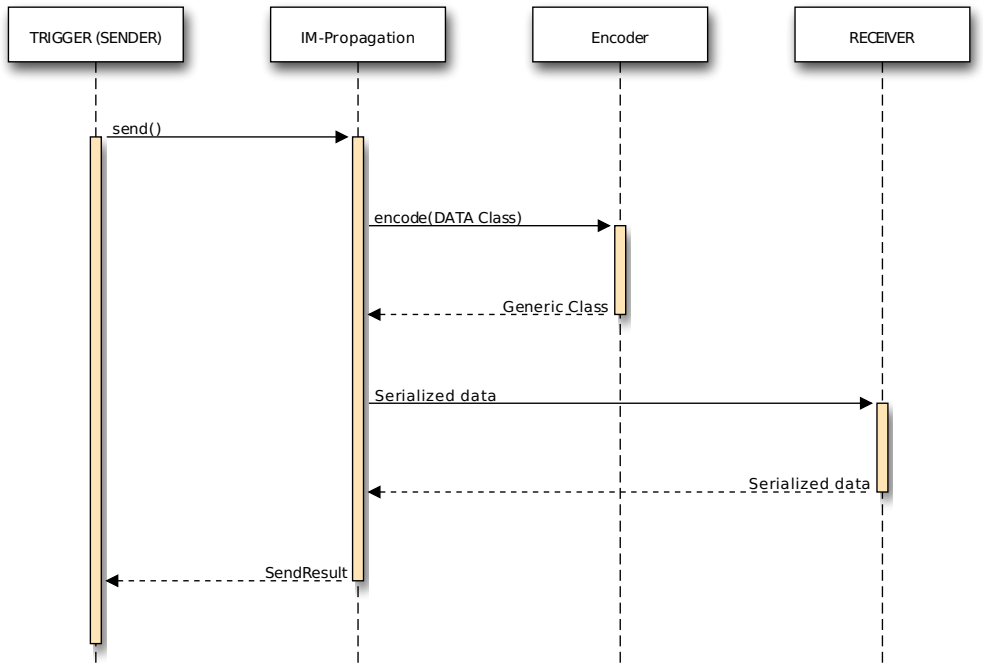

図 送信側の処理シーケンス

送信処理の手順

データを送信する場合は、PropagationManager#send() メソッドを使用します。

```
SampleData data = new SampleData("sample");
try (PropagationManager manager = PropagationManagerFactory.getInstance().getPropagationManager()) {
    manager.send(OperationType.DATA_CREATED, SampleData.class, data, EmptyObject.class);
} catch (final SendException e) {
    throw new RuntimeException(e);
}
```

表 引数一覧

引数	説明
第1引数	第1引数には、「データの操作種別」を指定します。 「データの操作種別」とは、渡すデータに対して送信側でどのような操作が行われたか、または、受信側に対してどのような操作を行って欲しいかを示す文字列です。 「データの操作種別」は「独自モデル」の完全修飾子（FQCN）ごとに送信側が自由に決めることができます。 IM-Propagation で、よく使用される操作種別があらかじめ登録されており、それを使用することもできます。一覧は、「*登録済みの「データの操作種別」一覧*」を参照してください。
第2引数	第2引数には、「独自モデル」のクラスを指定します。
第3引数	第3引数には、「独自モデル」のインスタンスを指定します。 ここで指定されたクラスのインスタンスは、「送信側のデータ変換クラス（Encoder）」に渡されます。
第4引数	第4引数には、受信側が返却した処理結果を受け取る場合に、データの受け口となるクラスを指定します。 特に処理結果が必要でない場合は、代わりに jp.co.intra_mart.foundation.propagation.model.EmptyObject のクラスを指定します。

第1引数（データの操作種別）と第2引数（「独自モデル」のFQCN）がキーとなり、「送信側のデータ変換クラス（Encoder）」と、後述する「受信側のデータ変換クラス（Decoder）」「受信側のデータ処理クラス（Procedure）」が IM-Propagation によって決定されます。
詳細は、「*マッピングの処理*」を参照してください。

データを送信する場合は、通常 IM-Propagation のセッション管理を使用する必要があります。
セッション管理についての詳細は、「*セッション管理*」を参照してください。

登録済みの「データの操作種別」一覧

IM-Propagation では、よく使用される「データの操作種別」があらかじめ登録されています。
登録されている「データの操作種別」は、以下の通りです。

表 データの操作種別一覧

キー名	概要	使用例
DATA_CREATED	データが新規作成された	自身の管轄するデータを新規作成したことを通知する目的で使用します。
DATA_UPDATED	データが更新された	自身の管轄するデータを更新したことを通知する目的で使用します。
DATA_DELETED	データが削除された	自身の管轄するデータを削除したことを通知する目的で使用します。
PROC_STARTED	処理が開始した	バッチなど何らかの処理が開始したことを通知する目的で使用します。
PROC_SUSPENDED	処理が一時停止した	バッチなど何らかの処理が一時停止したことを通知する目的で使用します。
PROC_RESUMED	処理が再開した	一時停止されていた処理が再開したことを通知する目的で使用します。
PROC_ABORTED	処理が中断した	実行中の処理が中断したことを通知する目的で使用します。
PROC_COMPLETED	処理が完了した	実行中の処理が完了したことを通知する目的で使用します。
PROC_FAILED	処理が失敗した	実行中の処理が失敗したことを通知する目的で使用します。
REQUEST_SEND	送信を要求	他のモジュールに対してメールなどの送信代行を依頼する目的で使用します。
REQUEST_COMMAND	実行を要求	他のモジュールに対してコマンドの実行代行を依頼する目的で使用します。
REQUEST_NOTIFY	通知を要求	他のモジュールに対して通知処理代行を依頼する目的で使用します。
REQUEST_SEARCH	検索を要求	他のモジュールに対して検索を依頼する目的で使用します。

受信側の処理

処理の流れ

受信側のデータ処理の流れは、以下の通りです。

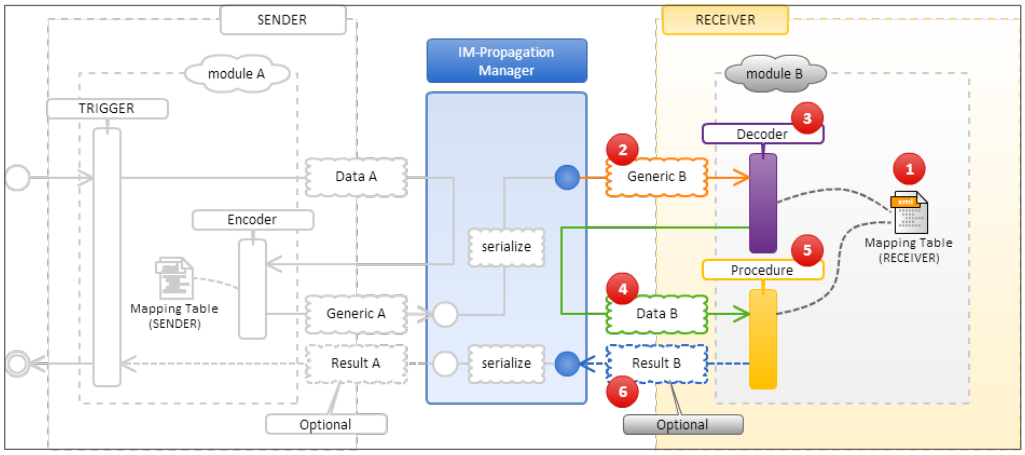

図 受信側の処理

1. データを受け取った IM-Propagation マネージャは、送られてきたデータを処理できる「受信側のデータ変換クラス（Decoder）」（図中の Decoder）「受信側のデータ処理クラス（Procedure）」（図中の Procedure）をマッピング設定（図中の Mapping Table）から決定します。
   マッピング設定からの決定方法については、「*マッピングの処理*」を参照してください。
2. IM-Propagation マネージャが、送信側から送られてきた「送受信モデル（Generic）」を、受信側が受け取る「送受信モデル（Generic）」（図中の Generic B）に変換します。
3. IM-Propagation マネージャが「受信側のデータ変換クラス（Decoder）」（図中の Decoder）のインスタンスを作成し、「送受信モデル（Generic）」のインスタンスを渡します。
4. 「受信側のデータ変換クラス（Decoder）」では、「送受信モデル（Generic）」のインスタンス内の情報を、「受信側のデータ処理クラス（Procedure）」が処理するためのデータを格納するためのクラス（以下、独自モデル。図中の Data B）に変換します。
5. IM-Propagation マネージャが「受信側のデータ処理クラス（Procedure）」（図中の Procedure）のインスタンスを作成し、「独自モデル」のインスタンスを渡します。
6. 送信側が何らかの処理結果を期待している場合は、処理結果を格納するクラス（図中の Result B）を用意して、IM-Propagation マネージャに処理結果を返します。
   処理結果を特に送らない場合は、不要です。

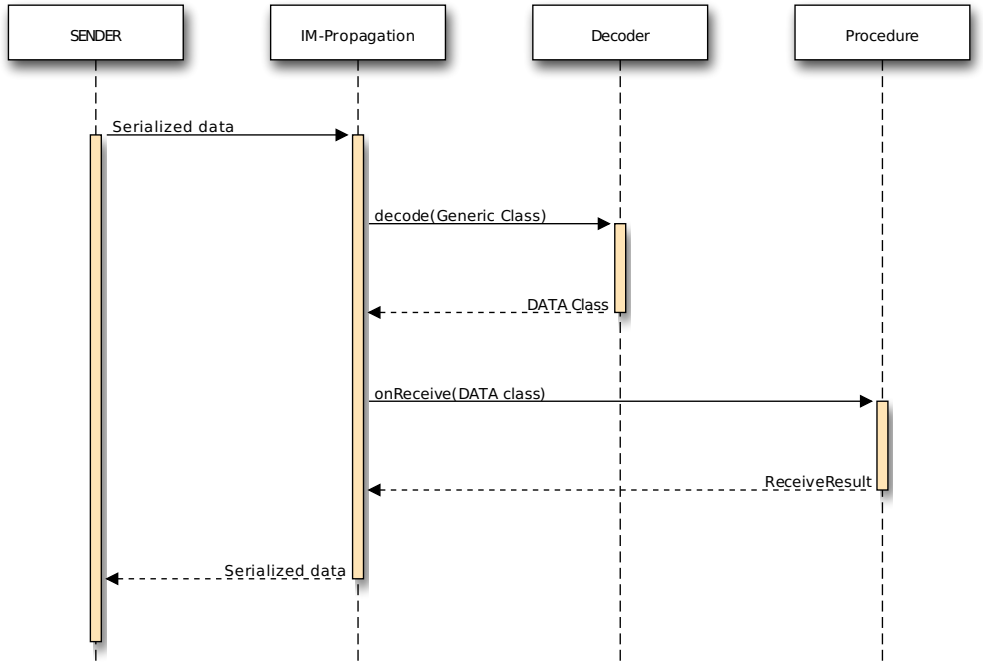

図 受信側の処理シーケンス

受信処理の手順

データを受信する場合は、「受信側のデータ処理クラス（Procedure）」に対して IM-Propagation からonReceive メソッドが呼び出され、その引数からデータを受け取ります。

```
public class SampleDataProcedure extends AbstractProcedure<GenericSampleData, EmptyObject> {
   @Override
   public ReceiveResult<EmptyObject> onReceive(final ReceiveParameter parameter,
         final GenericSampleData data) throws ProcedureException, PropagationManagerException {
      return new ReceiveResult<EmptyObject>(EventStatus.SUCCEEDED);
   }
}
```

表 引数一覧

引数/戻り値	説明
第1引数	第1引数から、「受信側のデータ処理クラス（Procedure）」が呼び出された状況を示す情報を取得できます。
第2引数	第2引数から、「受信側のデータ変換クラス（Decoder）」から渡された「受信側のデータ処理クラス（Procedure）」が処理するための「独自モデル」が取得できます。
戻り値	戻り値には ReceiveResult クラスのインスタンスを返却します。 この際、必ず結果ステータスを返却します。使用可能な結果ステータスの一覧は、「*返却可能な結果ステータス*」を参照してください。

通常は AbstractProcedure クラスを継承して「受信側のデータ処理クラス（Procedure）」を実装します。
ただし、データベース以外（ファイルやメモリなど）の操作を行う場合は、IM-Propagation のセッション管理を使用する必要があります。
セッション管理についての詳細は、「*セッション管理*」を参照してください。

返却可能な結果ステータス

IM-Propagation では、返却可能な結果ステータスがあらかじめ登録されており、それ以外のステータスは返却できません。
返却可能な「データの操作種別」は、以下の通りです。

表 データの操作種別一覧

キー名	概要	使用例
UNDEFINED	状態不明	IM-Propagation 内部で使用するステータスです。明示的には使用しません。
SUCCEEDED	処理が成功した	処理が成功した場合に返却します。
NOT_AFFECTED	処理は不要です	処理する必要がない場合に返却します。
NOT_IMPLEMENTED	未実装	処理すべきですが、現時点では未実装の場合に返却します。

キー名	概要	使用例
FAILED	処理が失敗した	処理が失敗した場合に返却します。

NOT_IMPLEMENTED または FAILED が返却された場合は、データの伝搬が失敗したとみなされ、例外が発生します。
この場合、onReceive メソッド内で ProcedureException を発生させた場合と同様の動作が行われます。

直列化（serialize）の処理

直列化の概要

送信側と受信側を動的に接続するために、送信側と受信側の間で行われるデータの伝搬を行う際に直列化（serialize）作業を行います。

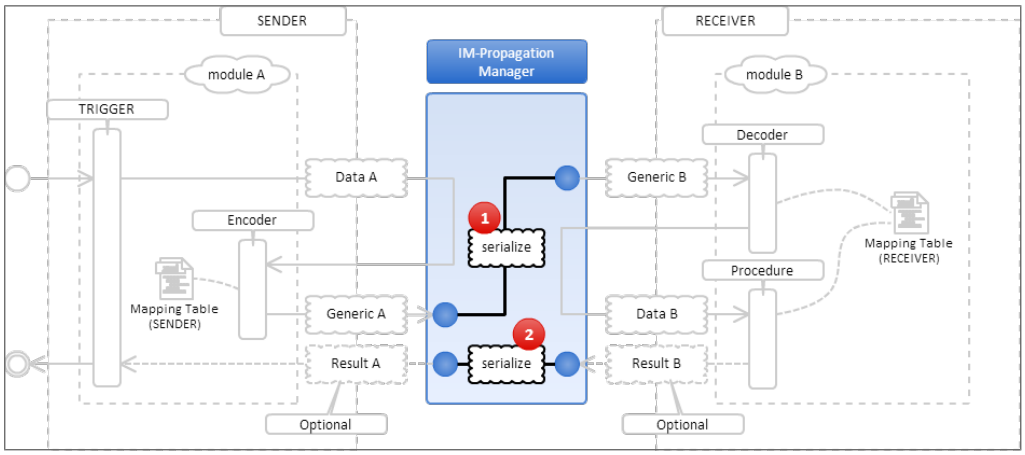

図 送受信間に流れるデータの直列化

1. 「送信側のデータ変換クラス（Encoder）」から「送受信モデル（Generic）」を受け取った IM-Propagation は、データを直列化して、「受信側のデータ変換クラス（Decoder）」が要求する「送受信モデル（Generic）」に変換します。
2. 「受信側のデータ処理クラス（Procedure）」から「処理結果を格納するクラス」を受け取った IM-Propagation は、データを直列化して、送信側が要求する「処理結果を格納するクラス」に変換します。

直列化によって、クラスや型情報が取り除かれ依存関係が切り離されるため、送信側と受信側の「送受信モデル（Generic）」「処理結果を格納するクラス」が必ずしも同一クラスを使用している必要はありません。
しかし、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。

直列化を行う際、クラス内のデータはプロパティごとに分割され記録されます。そして復元を行う際は、同じプロパティ名に対してデータが設定されます。

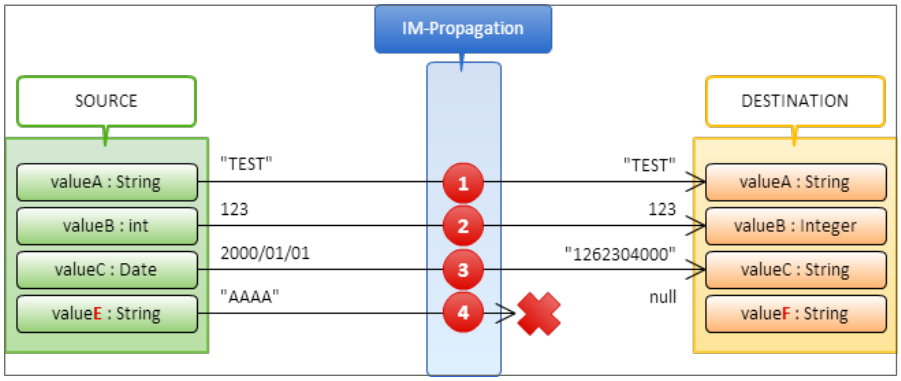

図 クラスの復元イメージ

1. 同じ型に対しては、まったく同じ内容が設定されます。
2. 異なる型でも Java 上で自動解決できる相互変換可能な型は、自動的に変換します。
3. 異なる型で自動的にキャストできない型の場合、「*変換ルール*」 に基づいて変換します。
4. 同じプロパティ名が受信側のクラスに存在しない場合は、値が設定されません（型のデフォルト値が設定されます）。

直列化をスムーズに行うために、「送信側のデータ変換クラス（Encoder）」と「受信側のデータ変換クラス（Decoder）」を使用して「送受信モデル（Generic）」の変換を行っており、「送受信モデル（Generic）」の実装には制約を持たせています。
「送受信モデル（Generic）」の制約は、以下の通りです。

- 総称型を使用していないこと（継承するクラス、実装するインタフェース、クラス内のプロパティを除く）。
- 単純な Getter/Setter で構成されたシリアライズ可能なクラスであること。
- 引数が 0 個のコンストラクタが用意されていること。

- 自身のクラスがプロパティの型に定義されているなどの理由で、親子関係が無限ループしていないこと。
- クラス内のプロパティの型が、プリミティブ型、または、伝搬機能内で直列化データと相互変換可能なクラス、または、インタフェースを実装したクラスに限定されていること（下記参照）。
  - 数値型
    - byte, java.lang.Byte
    - char, java.lang.Character
    - double, java.lang.Double
    - float, java.lang.Float
    - int, java.lang.Integer
    - long, java.lang.Long
    - short, java.lang.Short
    - java.lang.Number
    - java.math.BigDecimal
    - java.math.BigInteger
  - 真偽値型
    - boolean, java.lang.Boolean
  - 配列型
    - java.util.Collection
    - java.util.List
    - java.util.Map
    - java.util.Set
    - 固定長配列
  - その他
    - java.lang.Object
    - java.lang.String
    - java.net.URI
    - java.net.URL
    - java.util.Calendar
    - java.util.Date
    - java.util.Locale
    - java.util.TimeZone
    - java.util.UUID
    - その他、単純な Getter/Setter で構成されたシリアライズ可能なクラス

「処理結果を格納するクラス」は専用の「データ変換クラス」はありませんが、「処理結果を格納するクラス」自体に「送受信モデル（Generic）」と同じ制約を持たせることで、受信側から送信側方向へのデータ伝搬を実現しています。

変換ルール

入力と出力のクラスで、同じプロパティ名のプロパティに対して値が設定されます。
同じ型の場合は同じ値が渡され、異なる型の場合は以下の変換ルールに従って設定されます。

表 変換ルール一覧

入力の型	出力の型	変換ルール
byte[]	java.lang.String	BASE64 でエンコードした文字列に変換
java.util.Calendar	java.lang.String	1970年1月1日から起算したミリ秒に変換して文字列化
java.util.Date	java.lang.String	1970年1月1日から起算したミリ秒に変換して文字列化
java.lang.String 以外	java.lang.String	文字列に変換（byte[]、java.util.Calendar、java.util.Date を除く）
真偽値型	数値型	false のとき 0、true のとき 1
数値型	真偽値型	0 のとき false、それ以外のとき true
数値型	java.util.Calendar	1970年1月1日から起算したミリ秒として変換
数値型	java.util.Date	1970年1月1日から起算したミリ秒として変換
java.util.Calendar	数値型	1970年1月1日から起算したミリ秒に変換
java.util.Date	数値型	1970年1月1日から起算したミリ秒に変換

入力の型	出力の型	変換ルール
java.lang.String	真偽値型	空文字、"false"、"no"、"off"、"NaN" のとき false、それ以外のとき true
java.lang.String	数値型	文字列を数値に変換
java.lang.String	byte[]	BASE64 でエンコードした文字列とみなしてバイナリに変換
java.lang.String	java.net.URI	URI とみなして変換
java.lang.String	java.net.URL	URL とみなして変換
java.lang.String	java.util.Calendar	java.text.DateFormat で変換
java.lang.String	java.util.Date	java.text.DateFormat で変換
java.lang.String	java.util.Locale	ロケールIDとみなして変換
java.lang.String	java.util.TimeZone	タイムゾーンIDとみなして変換
java.lang.String	java.util.UUID	UUID とみなして変換
配列型	配列型	出力側の配列型に変換

上表に記載されていない型変換はサポートされておらず、データ伝搬時に例外が発生する可能性があります。

マッピングの処理

クラスの解決

「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）」「受信側のデータ処理クラス（Procedure）」を決定するために、送信側から指定された「独自モデル」の完全修飾子（FQCN）と、「データの操作種別」をキーにして、マッピング設定から選択します。

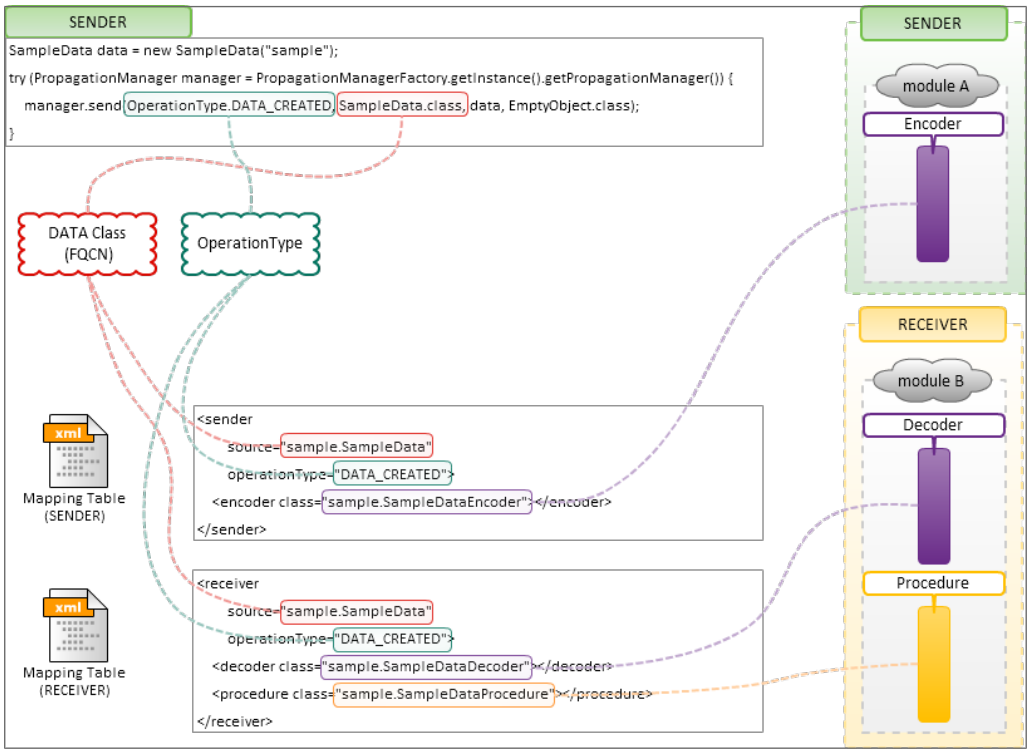

図 マッピング処理

マッピング設定は、送信側と受信側でそれぞれ用意し、設定ファイル（「IM-Propagation 送信側設定」と「IM-Propagation 受信側設定」）に定義します。
設定ファイルの詳細は、「設定ファイルリファレンス - IM-Propagation 送信側設定」「設定ファイルリファレンス - IM-Propagation 受信側設定」を参照してください。

データ変換クラスとデータ処理クラスの関係

「受信側のデータ処理クラス（Procedure）」は、データの処理単位で用意します。

例えば、2つの異なるモデルのデータを受け取っても最終的に同じデータ処理を行う場合、「受信側のデータ処理クラス（Procedure）」は1つだけ作成し、データ処理に最低限必要な共通の型を要求するようにします。
そして、異なる「送受信モデル（Generic）」から共通の型へ変換するために、それぞれのモデルに対応した「受信側のデータ変換クラス（Decoder）」（計2つ）を用意します。

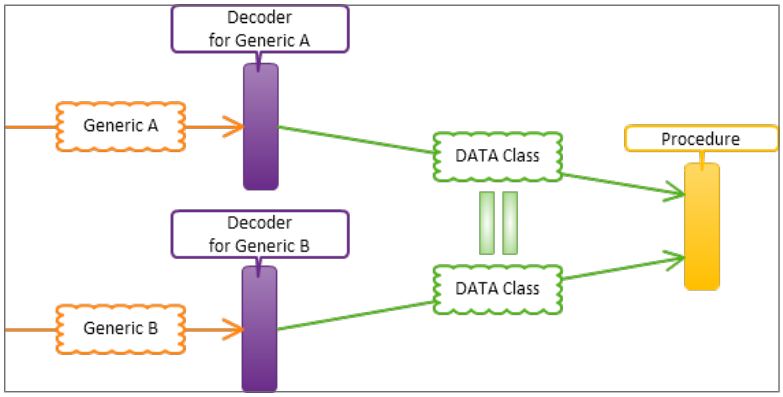

図 「受信側のデータ変換クラス（Decoder）」「受信側のデータ処理クラス（Procedure）」の関係

これにより、データ処理が同じで3つ目のモデルに対応する場合、3つ目のモデルから共通の型へ変換する「受信側のデータ変換クラス（Decoder）」を新しく作成するだけで対応できるようになります。
したがって、「受信側のデータ変換クラス（Decoder）」から「受信側のデータ処理クラス（Procedure）」に渡すクラスは、「送受信モデル（Generic）」ではなく受信側で用意した独自のクラスを渡すようにします。

セッション管理

項目

- セッション管理の概要
- 送信側の対処
- 受信側の対処

セッション管理の概要

IM-Propagation はデータの伝搬を行う際に、独自のセッション管理を行うことができます。
セッション管理を使用すると、IM-Propagation を使用してデータを伝搬する前後の処理で、以下の管理を同時に行います。

- データベースのトランザクション管理
- 「受信側のデータ処理クラス（Procedure）」に処理成功・失敗を通知するためのライフサイクルの管理

IM-Propagation のセッションを開始すると、自動的にデータベースのトランザクションが開始されます。
セッションを確定するとトランザクションがコミットされ、セッションを中断するとトランザクションがロールバックされます。

また、データの受信側がデータベース以外（ファイルやメモリなど）の操作を行っている場合に、途中でロールバックなどの処理を行うため、通常使用する `onReceive` 以外のセッション管理用のメソッドが呼び出されます。

通常、送信側は自身が送信するデータに対して、受信側がどのような処理を行うのかは予測できません。
したがって、送信側では IM-Propagation のセッション管理を使用するよう実装してください。
また、受信側では「データ処理クラス」で `onReceive` 以外のセッション管理用メソッドを実装してください。

送信側の対処

送信側で IM-Propagation のセッション管理を使用する場合は、`PropagationManager#begin()`、`PropagationManager#decide()`、および、`PropagationManager#abort()` メソッドを使用します。

```
SampleData data = new SampleData("sample");

final PropagationManager manager = PropagationManagerFactory.getInstance().getPropagationManager();
try {
   manager.begin();
   manager.send(OperationType.DATA_CREATED, SampleData.class, data, EmptyObject.class);
   manager.decide();
} finally {
   manager.abort();
}
```

1. まず最初に `begin` メソッドを呼び出して、セッションを開始します。
2. 次に通常通り `send` メソッドを呼び出して、データを送信します。
3. 最後に `decide` メソッドを呼び出して、セッションを確定して終了します。
4. 例外が発生した際にセッションを中断して終了するため、 `finally` ブロックで `abort` メソッドを呼び出します。

受信側の対処

受信側で IM-Propagation のセッション管理を使用する場合は、「データ処理クラス」で `AbstractProcedure` クラスの代わりに `AbstractSessionableProcedure` クラスを継承します。
`AbstractSessionableProcedure` クラスを継承することによって `onReceive` 以外のセッション管理用メソッドが IM-Propagation により呼び出され、各メソッド内でデータベース以外（ファイルやメモリなど）のコミット、および、ロールバック処理を行います。

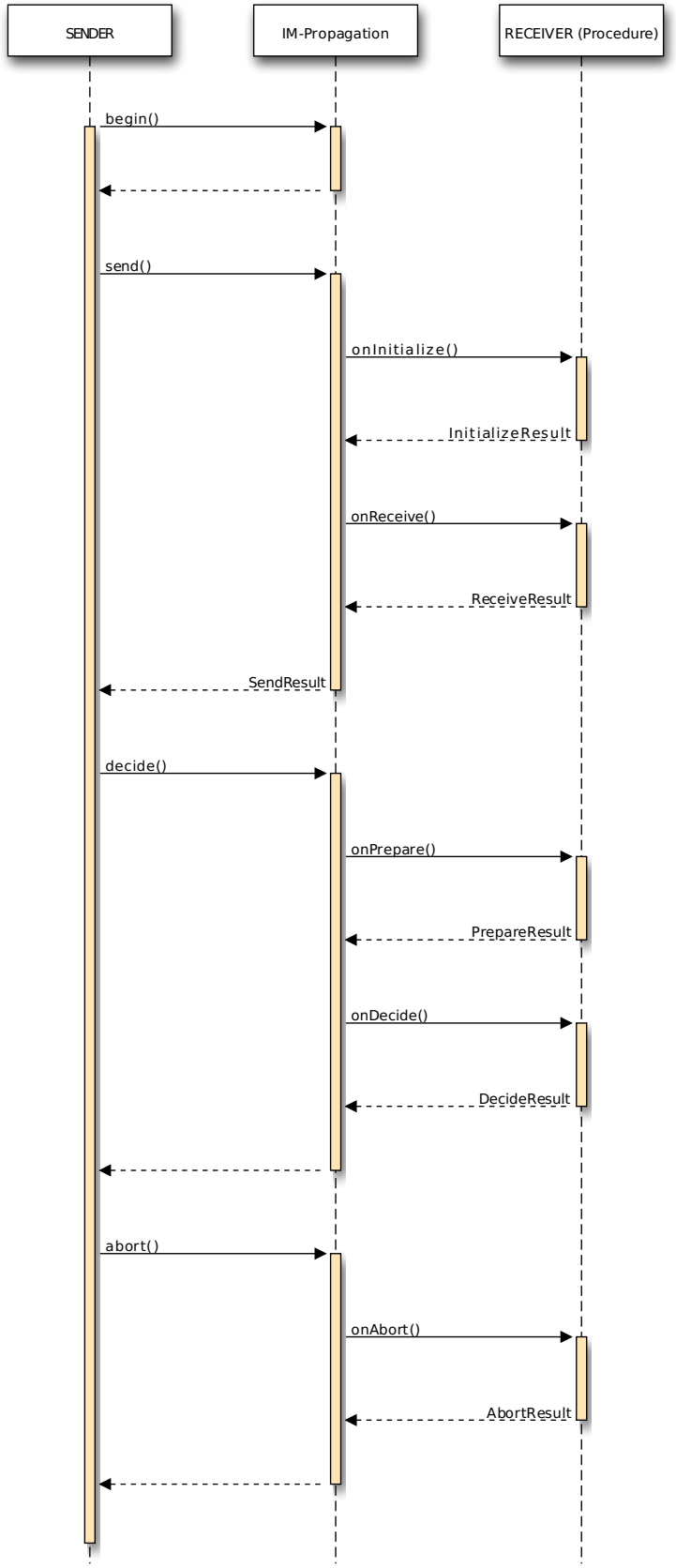

図 セッション管理使用時の処理シーケンス

IM-Propagation から呼び出されるメソッドの一覧は、以下の通りです。

表 メソッド一覧

メソッド名	概要	使用例
onInitialize	初期化処理	データ受信前の初期化処理を行います。 例えばファイルを操作する場合、変更対象のファイルを一時領域にコピーします。

メソッド名	概要	使用例
onReceive	受信処理	データの処理を行います。 例えばファイルを操作する場合、一時領域のファイルを更新します。
onPrepare	確定準備処理	セッションの確定を行う前の事前処理を行います。 例えばファイルを操作する場合、変更対象のファイルをロックします。
onDecide	確定処理	セッションの確定処理を行います。 例えばファイルを操作する場合、一時領域のファイルの内容で、変更対象のファイルを更新後、ロックを解除します。
onAbort	中断処理	セッションの中断処理を行います。 例えばファイルを操作する場合、一時領域のファイルを削除して、変更対象のファイルのロックを解除します。

注意

データベース以外の操作を行う場合は、全てのメソッドを処理して適切なセッション管理を独自に実装します。
正しくセッション管理を行わない場合、データの不整合の原因となります。

付録

項目
- デッドロック管理
  - デッドロック管理の概要
  - デッドロック時の動作

デッドロック管理

デッドロック管理の概要

IM-Propagation では、複数のスレッドから同一の「データ処理クラス」が実行されないようロック制御されています。
そのため、複数のスレッドでデータの伝搬が発生し、かつ、実行される「データ変換クラス」が連続する場合、デッドロックが発生する可能性があります。

IM-Propagation には、デッドロックが発生した場合の検出機能が組み込まれています。

「データ処理クラス」をロックする際の待機時間や、デッドロック検知時間の設定は、「IM-Propagation 設定」に定義します。
設定ファイルの詳細は、「設定ファイルリファレンス - IM-Propagation 設定」を参照してください。

デッドロック時の動作

デッドロックが発生する代表的なパターンを下図に示します。

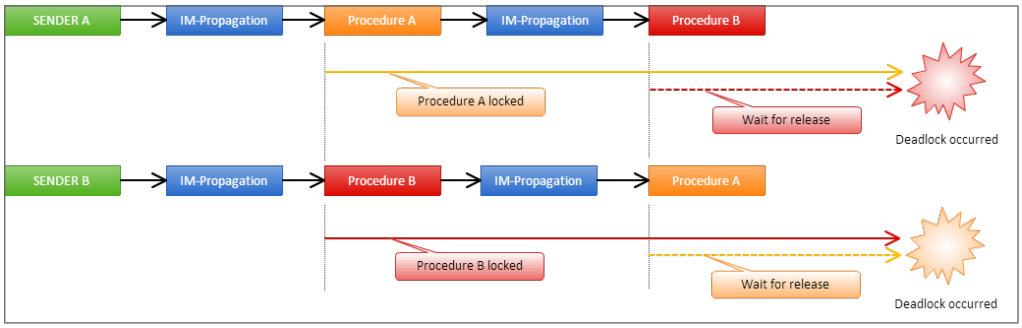

図 デッドロック発生時の例

デッドロックを検出すると、データの伝搬は失敗扱いとなり、PropagationManager#send() メソッドを実行した際に SendException が発生します。
送信側では、SendException を捕捉した場合、PropagationManager#abort() メソッドを呼び出してセッションの中断処理を行います。
すると、受信側にもセッションの中断処理を行わせるため、「データ処理クラス」の onAbort メソッドが呼び出されますので、後処理が必要な場合、このメソッド内で処理を行います。