Asakusa DSL 詳細設計および実装技法

改訂履歴

版	日付	内容
1.0	2012/01/17	新規作成

1. **はじめに** **4**

2. **詳細設計** **4**

2.1. バッチ詳細設計 4

2.2. ジョブフロー詳細設計 4

2.3. フローパート・演算子の詳細設計 5

2.3.1. エントリーポイントとして、入力データタイプの記述を行う.......... 5

2.3.2. エンドポイントとして、出力データタイプの記述を行う 5

2.3.3. 演算子情報.......... 5

3. **データ比較** **5**

3.1. データフローの実装方法 6

3.1.1. 変更されないフィールドを super-type に設定し、sub-type の処理を詳細に記述する方式6

3.1.2. 上記のようなケースで射影モデルを利用する方法.......... 7

3.1.3. 最小公倍数的なモデルを利用する方法 8

4. **演算子の割当** **8**

4.1. データ自体で CRUD 処理する場合 8

4.1.1. C : Create のケース 9

4.1.2. U : Update のケース 10

4.1.3. D : Delete のケース 11

4.1.4. R : Read のケース 12

4.2. 結合処理の場合 13

4.2.1. 単純結合.......... 13

4.2.2. 複雑な結合 15

4.2.3. 多数の種類のマスターや Tx と複雑な結合処理を行う場合 19

4.3. 集計処理の場合 20

4.4. グルーピング処理 23

4.5. フィールドの変形処理 25

5. **フロー制御の割当** **26**

1. はじめに

基本設計で作成された DAG から実装を行うまでの方針と具体的な手順を示す。ここでは、DAG で構成された基本設計から詳細設計を行い、実際に実装に展開する。 AsakusaDSL 設計手法を事前に読むことを推奨する。

まず DSL の構成について述べる。
AsakusaDSL は下記の図の通り多層的な 3 種類の DSL で構成されている。詳細設計はこのそれぞれの DSL でのコードの記述を行うための設計を行う。

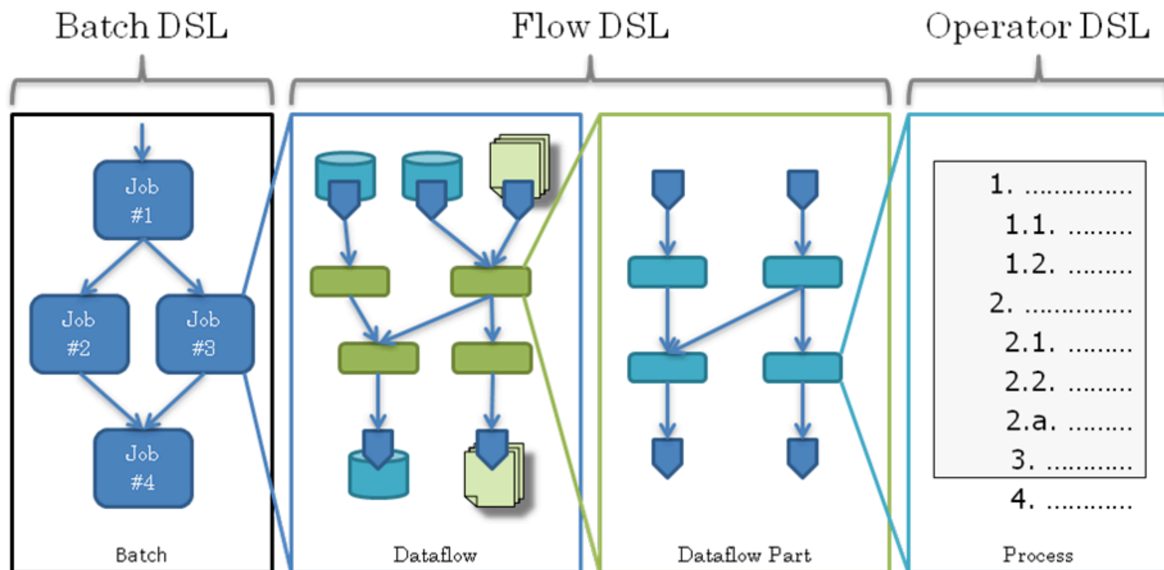

図 1

基本設計で DAG は既に構成されているので、この詳細設計ではその DAG に対する演算子（OperatorDSL）の割当とその実装、および上位のフロー・フローパーツ(FlowDSL)、ならびにバッチ(BatchDSL)の実装の詳細を決定する。

2. 詳細設計

詳細設計を、要件定義->基本設計->詳細設計->実装->テスト->リリースという開発フローの中で、基本設計を入力とし、適切な詳細設計書を出力するフェーズとする。基本設計では、バッチ一覧・ジョブフロー一覧・ジョブフロー設計書を作成する。ジョブフロー設計書の作成で DAG・DAG の説明・構成するフローパートの記述等を行う。これをもとに詳細設計を行う。作成される詳細設計書は実装のために以下の必要な情報を含む。

2.1. *バッチ詳細設計*

バッチクラスの記述の為に必要な項目を設計する。バッチは最上位のコンセプトであり、できるだけ一覧性を優先させる。基本設計書のバッチ一覧とジョブフロー一覧の記述を入力とする。バッチの詳細設計では、バッチを構成するジョブフローの一覧から、ジョブフローの具体的な構成と処理の順序を詳細に設計していく。

ジョブフローの一覧で表示されている各ジョブフローに対して、ジョブフローのクラス名を対応させる。また管理情報として、バッチ名称・管理名称・クラス名・管理番号・管理コードを設定する。バッチ DSL の記述に利用する。

2.2. *ジョブフロー詳細設計*

Copyright 2012 Asakusa Framework Team All Rights Reserved.

ジョブフロークラスの記述の為に必要な項目を設計する。基本設計で作成された DAG ベースの基本設計書をベースに、以下のジョブフロークラスの項目を設計する。

・クラス名：ジョブフロー管理項目の名称・ID と対応させる
・入出力の型：インプットデータモデルとアウトプットデータモデルを設計した DDL のクラス名や ID 等の管理項目を設定する。
・内部の結線：利用する演算子と構成フローパートのクラス名・ID、結線するデータモデル・演算子・メソッド名を決定・記述する。以降のフローパート・演算子の詳細設計により構成される。

2.3. *フローパート・演算子の詳細設計*

具体的なフローパート・演算子の詳細を設計する。具体的にコードにおとすフォーマットに記述する。フォーマットは以下の例に従う。

2.3.1. エントリーポイントとして、入力データタイプの記述を行う

フィールド名と型を記述する。また、このデータインスタンスのユニークネスを保証するキーを決める。また入出力のデータモデルは必要なネスト構造がわかるように記述しておくと後で便利である。

2.3.2. エンドポイントとして、出力データタイプの記述を行う

フィールド名と型を記述する。フローパートとの整合性がとれるように、途中のフローパートや入力モデルとの関連を記述しておく。

2.3.3. 演算子情報

演算子のシーケンスな並びと、各演算子が入力データタイプのどのフィールドにどのような Read/Update/Delete 操作を行い、出力データタイプのどのフィールドに対して、どのような Create 操作を行うか、を記述する。また、データのフローが分岐している場合に適当なフロー制御演算子の割当を記述する。

以上の詳細設計書をもとに、実装を行う。またテスト計画書を作成し、単体テスト・結合テストを実施する。

具体例についてはチュートリアルを参照にすること。

[演算子・フローパートの設計・選択方針]
以降、与えられた DAG から演算子・フローパートを設計・実装する方法について述べる。ジョブフローを具体的に設計・実装することができる。

全体の流れとして、エントリーポイントとエンドポイントのデータ比較から、適切な演算子を抽出していく。一般にジャクソン法と言われる手法を一部援用する。以降 Asakusa の Operater/DMDL に依存する。

3. データ比較

入出力のデータタイプを比較して、変更されているフィールドと構造に注目する。原則としてフィールドの操作がアトミックである単位で演算子を割り当てて行く。一般に言われる CRUD モデルを参考にし、データフローを意識した CRUD のフローを作成していく。（CRUD モデルは、各フィールドに対する操作を C:Create(生成） R:Read（読み込み） U:Update（更新） D:Delete（削除）に分類する手法である。）

参考として下記に CRUD のフロー図を掲示する。各行が各フィールドへの一連の操作になっている。基本的にそのフィールドに対して操作を行う内容が、対応する Operator のカラムになっている。各行を順に追っていくと、特定のフィールドに対する操作が一覧でき、各列を見ることで、どの演算子がどのフィールドにどんな操作しているかを一覧することができる。

Copyright 2012 Asakusa Framework Team All Rights Reserved.

DAG との対応関係で述べると、エッジが入力データモデルのフィールドと出力データモデルのフィールドになり、頂点の処理が Operator で構成されている。下記の例では、5 種類の Operator で一つの頂点が形成されていることになる。

入力データモデルメンバー	operator1	operator2	operator3	operator4	operator5	出力モデルメンバー
a	R	→	R	→	→	a
b	→	→	→	R	→	b
c	R	R	D			
d	→	→	→	→	→	d
e	→	→	→	→	R	e
f	→	→	U	→	→	f
			C	→	U	g

基本的にデータ構造として、変化しないフィールド(R/copy(->))と処理の過程で変更されるフィールド(C/U/D)に分けて認識しておくとよい。

3.1. *データフローの実装方法*

上記データフローにおけるデータモデルの実装方法は大きく以下の 3 通りある。

3.1.1. 変更されないフィールドを super-type に設定し、sub-type の処理を詳細に記述する方式

見通しがよくなる。特にデータ・フィールドの数が多い場合には有効な手法になる。データ構成が複雑ではない場合は特段、利用する必要はない。CRUD のフロー図を例示する。

	入力データモデルメンバー	operator1	operator2	operator3	operator4	operator5	出力モデルメンバー
superType	a	R	→	R	→	→	a
	b	→	→	→	R	→	b
	d	→	→	→	→	→	d
	e	→	→	→	→	R	e
subType	c	R	R	D			
	f	→	→	U	→	→	f
				C	→	U	g

SuperType を利用した例を示す。MasterJoinUpdate でサブタイプの部分だけ更新するような場合は、フローパートで wrapper をつくるとよい。

[DMDL]

```
super_type = {
    order_type : INT;
    item_code : INT;
    quantity : INT;
};
sub_type = super_type + {
    selling_price : INT;
};
item_info = {
    item_code : INT;
    unit_price : INT;
};
```

Copyright 2012 Asakusa Framework Team All Rights Reserved.

```
[ Operator DSL ]
public abstract class JoinOperator {
    @MasterJoinUpdate
    public void updateSubType(
            @Key(group = "item_code") ItemInfo itemInfo,
            @Key(group = "item_code") SubType subtype) {
        subtype.setSellingPrice(subtype.getQuantity() * itemInfo.getUnitPrice());
    }
}
```

[Flow DSL]

```
private In<SuperType> superType;
private Out<SubType> subType;
@Override
protected void describe() {
    // フィールドを拡張して元のフィールドをコピーする。
    CoreOperatorFactory core = new CoreOperatorFactory();
    Extend<SubType> extended = core.extend(superType, SubType.class);
    // 商品マスタと結合して追加したフィールドを操作する。
    // join oprerator の中で superType のフィールドにアクセスもできる上に隠蔽もできる。
    JoinOperatorFactory joinOp = new JoinOperatorFactory();
    UpdateSubType out = joinOp.updateSubType(extended);

    subType.add(out);
}
```

3.1.2. 上記のようなケースで射影モデルを利用する方法

SuperType をフロー全体に対する射影モデルとして定義し、その拡張モデルの SubType で射影モデルのフィールドに対してデータ操作を行う。
拡張モデルが複数定義されていても射影モデルのフィールド操作を共通の処理として利用することができる。

[DMDL]

```
projective super_type = {
    order_type : INT;
    order_code : INT;
    selling_price : INT;
};
total_type = super_type + {
    total : INT;
};
tax_type = super_type + {
    tax : INT;
};
```

・共通のフィールドのみを処理する演算子の記述。射影モデルの SuperType を拡張したモデルで共通に利用することができる。

[Operator DSL]

```
     @Update
    public <E extends SuperType> void setOrderType(E subType, int orderType) {
        subType.setOrderType(orderType);
    }
```

Copyright 2012 Asakusa Framework Team All Rights Reserved.

[Flow DSL]
```
private In<TotalType> totalType;
private Out<TotalType> out;
@Override
protected void describe() {
    // 拡張したモデルで射影モデルのフィールドを操作することができる。
    out.add(operator.setOrderType(totalType,0).out);
}
```

3.1.3. 最小公倍数的なモデルを利用する方法

基本的にフィールド追加・削除されるたびに型が変更されることになる。これはコンパイラによるチェックを最大限に利用するためである。ただしアトミックな業務処理で分割可能性が低い場合は、入出力 IO の記述を減らすために、入力時に一気にタイプ変換を行い、最小公倍数的な SuperType をつくっておき、連続的に処理するということも可能である。ただしこの場合は「途中の処理が変更された」時に変更にコストがかかるので、必ず比較考量すること。

・DMDL の記述例
最初にすべてのフィールドを含む Type で記述する。

```
type = {
    code : INT;
    name : TEXT;
    count : INT;
};
```

・演算子の記述～単純に Update だけ呼び出す演算子で extend を利用しない
[Operator DSL]
```
     @Update
    public void addCount(Type type) {
        type.setCount(type.getCount() + 1);
    }
```

[Flow DSL]
```
In<Type> type;
Out<Type> out;
protected void describe() {
    AddCount updated = operator.addCount(type);
    out.add(updated.out);
}
```

4. 演算子の割当

入出力の比較で得られた、変更フィールドに対する CRUD に対して、適切な演算子を割り当てていく。とくに Create/Update については適切な演算子を割り当てる際に注意が必要である。演算子の割当方針は以下を参考にする。

4.1. *データ自体で CRUD 処理する場合*

上記のデータ比較で、各項目に CRUD が設定されている場合で、結合等の処理を行わない場合は以下の演算子を選択する。

Copyright 2012 Asakusa Framework Team All Rights Reserved.

4.1.1. **C : Create のケース**

Convert / Extend / Restructure を選択する。

・全く新しいデータタイプを作る場合は Convert を利用する。
データ項目を一つ追加するケースで、例えば合計値をセットするフィールドを新たに作って、計算結果を格納する例で型を変更するケースは以下の通りになる。

[DMDL]

```
before_type = {
  code : INT;
  quantity : INT;
  selling_price : INT;
};

after_type = {
  code : INT;
  quantity : INT;
  selling_price : INT;
  total : INT;
};
```

[Operator DSL]

```
public abstract class ConvertOperator {
    private AfterType after = new AfterType();

   @Convert
    public AfterType toAfterType(BeforeType before) {
        after.setCode(before.getCode());
        after.setSellingPrice(before.getSellingPrice());
        after.setQuantity(before.getQuantity());
        after.setTotal(before.getSellingPrice() * before.getQuantity());
        return after;
    }
}
```

[Flow DSL]

```
private CoreOperatorFactory core = new CoreOperatorFactory();
private ConvertOperatorFactory operator = new ConvertOperatorFactory();
private In<BeforeType> beforeType;
private Out<AfterType> afterType;

@Override
protected void describe() {
    // SuperType 型から、SubType 型に変換する。
    ToAfterType converted = operator.toAfterType(beforeType);
    afterType.add(converted.out);
    // 変換前のデータの出力先は停止にする。
    core.stop(converted.original);
}
```

・既存のデータタイプにフィールドを追加する場合は Extend を利用する。
上記を同じケースで、型を拡張するケースがこれにあたる。

[DMDL]

```
 super_type = {
    code : INT;
```

Copyright 2012 Asakusa Framework Team All Rights Reserved.

```
    quantity : INT;
    selling_price : INT;
};
sub_type = super_type + {
    total : INT;
};
```

[Operator DSL]

```
public abstract class SetTotalOperator {
     @Update
    public void setTotal(SubType subType) {
        subType.setTotal(subType.getSellingPrice() * subType.getQuantity());
    }
}
```

[Flow DSL]

```
In<SuperType> superType;
Out<SubType> subType;
@Override
private SetTotalOperatorFactory operator = new SetTotalOperatorFactory();
protected void describe() {
    // total フィールドを追加した SubType 型へ変換する。
    Extend<SubType> extended = core.extend(superType, SubType.class);
    // total フィールドに計算結果を設定する。
    SetTotal updated = operator.setTotal(extended);
    subType.add(updated.out);
}
```

・既存のデータタイプを利用して、新しいデータタイプを作る場合は Restructure を利用する
基本的に、Extend と同じになるが、演算子が異なる。尚、Restructure はコンパイラによる型チェックが甘くなるので、Convert/Extend を推奨する。

[Flow DSL]

```
In<SuperType> superType;
Out<SubType> subType;
@Override
private SetTotalOperatorFactory operator = new SetTotalOperatorFactory();
protected void describe() {
    // total フィールドを追加した SubType 型へ変換する。
    Restructure<SubType> restructured = core.restructure(superType, SubType.class);
    // total フィールドに計算結果を設定する。
    SetTotal updated = operator.setTotal(restructured);
    subType.add(updated.out);
}
```

4.1.2. U : Update のケース

Update を選択する。

フィールドの更新処理を行う場合はこの演算子を利用する。
例えば、もともとも合計値のフィールドを持っている型に対して、合計値を計算して更新処理するケースは以下のようになる。

Copyright 2012 Asakusa Framework Team All Rights Reserved.

[Operator DSL]
```
     @Update
    public void addTotalValue(Bar bar) {
        bar.setTotal(bar.getQuantity() * bar.getSellingPrice());
    }
```

4.1.3. D : Delete のケース

Convert / Project / Restructure を選択する。

・フィールドを削除して、全く新しいデータタイプを作る場合。Convert を利用する。合計値を格納するフィールドを除去する例。

[DMDL]
```
before_type = {
    code : INT;
    selling_price : INT;
    total : INT;
};

after_type = {
    code : INT;
    selling_price : INT;
};
```

[Operator DSL]
```
public abstract class ConvertOperator {
    private AfterType after = new AfterType();
    @Convert
    public AfterType toAfterType(BeforeType before) {
        after.setCode(before.getCode());
        after.setSellingPrice(before.getSellingPrice());
        return after;
    }
}
```

[Flow DSL]
```
private ConvertOperatorFactory converter = new ConvertOperatorFactory();
private In<BeforeType> beforeType;
private Out<AfterType> afterType;

@Override
protected void describe() {
    // SubType 型から、SuperType 型に変換する。
    ToAfterType converted = converter.toAfterType(beforeType);
    afterType.add(converted.out);
    // 変換前のデータの出力先は停止にする。
    core.stop(converted.original);
}
```

・既存のデータタイプからフィールドを削除する場合は Project を利用する
上記のケースで型を射影するケースがこれにあたる。以下のようになる。

[DMDL]

Copyright 2012 Asakusa Framework Team All Rights Reserved.

```
super_type = {
    code : INT;
    selling_price : INT;
};
sub_type = super_type + {
    total : INT;
};
```

[Flow DSL]

```
private In<SubType> subType;
private Out<SuperType> superType;

@Override
protected void describe() {
    CoreOperatorFactory core = new CoreOperatorFactory();
    // SubType 型の total フィールドを除去して、SuperType 型に変換する。
    Project<SuperType> projected = core.project(subType, SuperType.class);
    superType.add(projected);
}
```

・既存のデータタイプを利用して、新しいデータタイプを作る場合は Restructure を利用する。上記のケースでは、基本的に Convert と同じになる。演算子が異なるだけである。Restructure はコンパイラによる型チェックが甘くなるので、Convert/ Project を推奨する。

[Flow DSL]

```
private In<SubType> subType;
private Out<SuperType> superType;

@Override
protected void describe() {
    CoreOperatorFactory core = new CoreOperatorFactory();
    // SubType 型の total フィールドを除去して、SuperType 型に変換する。
    Restructure<SuperType> restructured = core.restructure(subType, SuperType.class);
    superType.add(restructured);
}
```

4.1.4. R : Read のケース

特に特別の演算子は必要ない。普通にフィールドに get するだけでよい。下の例は、コードをキーにしてグルーピングし、順にそのコードにアクセスするケース。

[DMDL]

```
super_type = {
    code : INT;
    quantity : INT;
    selling_price : INT;
    total : INT;
};
```

[Operator DSL]

```
public abstract class ExtractOperator {
```

Copyright 2012 Asakusa Framework Team All Rights Reserved.

```
    @Update
    public void extract(@Key(group = "code") SuperType superType) {
        // SuperType のフィールドを get する。
        superType.setTotal(superType.getQuantity() * superType.getSellingPrice());
    }
}
```

上記はデータモデル内でのデータ変形であり、特に外部との連携はしないケースである。Create/Delete の場合は型変更になるケースが多いので、Convert を利用することが多い。ただし、コードの見通しを上げるのであれば、Project 等を利用する。

4.2. *結合処理の場合*

大抵の場合、Create/Update の処理では結合処理が行われることが多い。Read で結合キーを取得して、結合処理が行われ、フィールドが追加または修正されるというケースがこれに相当する。結合処理は、単純結合と複雑な結合に分けて考える。

4.2.1. 単純結合

単純に、あるデータと別のデータのそれぞれのフィールドを一つのキーで等価結合する場合である。この種の単純結合の場合は、手法が二種類ある。

一つは DMDL で記述を済ませてしまう方法である。データモデルの中だけで処理を記述してしまう。この場合は演算子の内部の実装は必要ない。結線に必要な抽象クラスを記述するだけある。抽象クラスで利用する演算子は MasterJoin と Split になる。実装は DMDL での結合クラスでの記述になる。

[単純結合の例：DMDL を利用したケース]
MasterJoin 利用の具体例として、商品マスターとトランザクションデータの結合処理を記述する。商品コードをキーにして、商品名を取得して、名前をセットするケースである。

[DMDL]
```
"商品マスタ"
item_info = {
    item_code : TEXT;
    item_name : TEXT;
};
"売上明細"
sales_detail = {
    item_code : TEXT;
    amount : INT;
    selling_price : INT;
};
"売上明細+商品マスタ"
joined joined_sales_info =
    sales_detail % item_code +
    item_info % item_code ;
```

尚、結合後の joined_sales_info は、item_info と sales_detail の両者のフィールドをすべて持つ。（不要なフィールドを削除することもできる。詳しくは、DMDL ユーザーガイドを参考にすること[1]）結合後に結合前のモデルを利用したい場合は後述する Split を用いる。

[1] http://asakusafw.s3.amazonaws.com/documents/0.2/release/ja/html/dmdl/user-guide.html

Copyright 2012 Asakusa Framework Team All Rights Reserved.

以下、結合して処理をそのまま流すケース
[Operator DSL]

```
@MasterJoin
public abstract JoinedSalesInfo joinItemInfo(ItemInfo info, SalesDetail sales);
```

[Flow DSL]

```
In<SalesDetail> salesDetail;
In<ItemInfo> itemInfo;
Out<JoinedSalesInfo> joinedSalesInfo;
protected void describe() {
    // 商品マスタと売上明細を結合する
    JoinItemInfo joinItemInfo = operator.joinItemInfo(itemInfo, salesDetail);
    // 結合結果をフローの出力に結線する
    joinedSalesInfo.add(joinItemInfo.joined);
    // 結合に失敗した場合は停止する（出力しない）
    core.stop(joinItemInfo.missed);
}
```

尚、結合後に、結合前のモデルの分離して処理を続行する場合は Split を利用する。
[Operator DSL]

```
@Split
public abstract void split(JoinedSalesInfo joined,
        Result<SalesDetail> salesDetail,
        Result<ItemInfo> itemInfo);
```

[Flow DSL]

```
In<JoinedSalesInfo> joinedSalesInfo;
Out<SalesDetail> salesDetail;
Out<ItemInfo> itemInfo;
protected void describe() {
    // 結合モデルから元のレコードに分割する
    Split split = operator.split(joinedSalesInfo);
    salesDetail.add(split.salesDetail);
    itemInfo.add(split.itemInfo);
}
```

二つ目は意図的に演算子で記述するケースである。結合と同時に何か別の処理をしたい場合などに選択される。演算子は MasterJoinUpdate が選択される。演算子で記述するので、細かい処理も追記できる。

[単純結合の例：演算子で記述したケース]
MasterJoinUpdate の記述例として、商品マスターとトランザクションデータの結合処理を記述する。商品コードをキーにして、商品名をセットして、更新完了フラグをセットするような場合は以下のようになる。

更新後のフィールドを持つ拡張モデルを定義する。
[DMDL]

```
"商品マスタ"
item_info = {
    item_code : TEXT;
    item_name : TEXT;
};

"売上明細"
sales_detail = {
    item_code : TEXT;
    amount : INT;
    selling_price : INT;
```

Copyright 2012 Asakusa Framework Team All Rights Reserved.

```
};

“売上明細”の拡張
sales_info = sales_detail + {
    item_name : TEXT;
    update_flg : BOOLEAN;
} ;
```

[Operator DSL]

```
    @MasterJoinUpdate
    public void updateItem(
            @Key(group = “itemCode”) ItemInfo itemInfo,
            @Key(group = “itemCode”) SalesInfo salesInfo) {
        // 更新完了フラグと商品名を拡張モデルにセットする
        salesInfo.setUpdateFlg(true);
        salesInfo.setItemName(itemInfo.getItemName());
    }
```

[Flow DSL]

```
    In<SalesDetail> salesDetail;
    In<ItemInfo> itemInfo;
    Out<SalesInfo> salesInfo;
    protected void describe() {
        // 更新後のフィールドを持つ拡張モデルに変換する
        Extend<SalesInfo> extended = core.extend(salesDetail, SalesInfo.class);
        // 商品マスタと売上明細を結合する
        UpdateItem updateItem = operator.updateItem(itemInfo, extended);
        // 結合結果をフローの出力へ結線する
        salesInfo.add(updateItem.updated);
        // 商品マスタが見つからなかった場合は停止する（出力しない）
        core.stop(updateItem.missed);
    }
```

4.2.2. 複雑な結合

複雑な結合処理がしたい場合は演算子処理の記述を選択する。処理の内容によって利用する演算子が異なる。以下個別のケースについて例示していく。

[存在チェックを行う場合の結合処理]

特に基幹処理バッチでは頻出する処理である。データクレンジングではよく行われる処理で、トランザクションデータのセマンティクス・バリデーションに多用される。「そのデータは存在していいのですか？」というような確認に利用する。

基本的に MasterCheck を利用する。例外の場合は自動的に結線の出力先を分けてくれる。（MasterBranch では例外処理は NULL 出力になるので、存在チェックには利用しない方がよい。）

存在チェックの例として、トランザクションデータ（売上明細）に対して、顧客マスタの存在チェックを行うケースを例示する。結合に失敗した場合のエラー処理は、別のデータフローに流す処理になる。

[DMDL]

```
“顧客マスタ”
customer_info = {
    customer_code : TEXT;
    customer_name : TEXT;
    valid_flg : BOOLEAN;
    valid_date : DATE;
};
“売上明細”
```

```
sales_detail = {
    customer_code : TEXT;
    amount : INT;
    selling_price : INT;
};
```

[Operator DSL]

```
    @MasterCheck
    public abstract boolean existsCustomer(
            @Key(group = "customerCode") CustomerInfo customerInfo,
            @Key(group = "customerCode") SalesDetail salesDetail);
```

[Flow DSL]

```
    In<SalesDetail> salesDetail;
    In<CustomerInfo> customerInfo;
    Out<SalesDetail> found;
    Out<SalesDetail> error;
    protected void describe() {
        ExistsCustomer exists = operator.existsCustomer(customerInfo, salesDetail);
        // 顧客マスタが存在した場合
        found.add(exists.found);
        // 顧客マスタが存在しなかった場合
        error.add(exists.missed);
   }
```

[selection を利用した存在チェックの例]

上記の例で、さらに顧客マスターのうち利用可能フラグ（valid_flg）が真になっているものだけで、結合処理をしたいような場合はマスタ選択（selection 演算子）を利用する。

[Operator DSL]

```
    @MasterCheck(selection = "selectValidCustomer")
    public abstract boolean existsCustomer(
            @Key(group = "customerCode") CustomerInfo customerInfo,
            @Key(group = "customerCode") SalesDetail salesDetail);

    @MasterSelection
    public CustomerInfo selectValidCustomer(List<CustomerInfo> customers) {
        for (CustomerInfo customerInfo : customers) {
            // 利用可能フラグの判定
            if (customerInfo.getValidFlgOption()) {
                return customerInfo;
            }
        }
        return null;
    }
```

[BatchContext を利用した存在チェックの例]

上記の例で、さらに顧客マスターのうち利用可能フラグが真になっているものに加えて、有効日付（valid_date）がバッチ実行時に決定する指定日以降を対象として、結合処理をしたいような場合は selection 演算子を利用した上で、BatchContext を利用する。

[Operator DSL]

```
    @MasterSelection
    public CustomerInfo selectValidCustomer(List<CustomerInfo> customers) {
        // バッチパラメータから値（YYYYMMDD の文字列表現）を取得する。
        String dateAsString = BatchContext.get("VALID_DATE");
        // 文字列表現から com.asakusafw.runtime.value.Date 型へ変換する。
        Date validDate = Date.valueOf(dateAsString, Format.SIMPLE);
        for (CustomerInfo customerInfo : customers) {
```

```
            if (customerInfo.getValidFlgOption.get() &&
                customerInfo.getValidDate().getElapsedDays() >= validDate.getElapsedDays()) {
                return customerInfo;
            }
        }
        return null;
    }
```

[結合処理時に追加的に複雑な操作を行う場合]
トランザクションデータとマスターを結合して、複雑な処理を追加的に行うケースについて述べる。

結合して処理を分岐させたいときは MasterJoin で結合した後に Branch を利用する。例えば、トランザクションデータと商品マスターを結合させて、条件によって処理を分岐させるようなケースがこれにあたる。商品マスターと売上明細を商品コードで結合して、カテゴリーコード別に処理を振り分けるようなケースは以下のように記述する。

[DMDL]

```
"商品マスタ"
item_info = {
    item_code : TEXT;
    item_name : TEXT;
    category_code : TEXT;
};

"売上明細"
sales_detail = {
    item_code : TEXT;
    selling_price : INT;
   amount : INT;
    };

"売上明細+商品マスタ"
joined joined_sales_info =
    item_info % item_code +
    sales_detail % item_code
    ;
```

[Operator DSL]

```
    public enum ResultType {
        CATEGORY_A,
        CATEGORY_B,
        UNKNOWN;
    }
    @MasterJoin
    public abstract JoinedSalesInfo joinItemInfo(ItemInfo info, SalesDetail sales);

    @Branch
    public ResultType separate(JoinedSalesInfo salesInfo) {
        if (salesInfo.getCategoryCode().equals("01")) {
            return CATEGORY_A;
        } else if (salesInfo.getCategoryCode().equals("02")) {
            return CATEGORY_B;
        } else {
            return UNKNOWN;
        }
    }
```

[Flow DSL]

```
    In<SalesDetail> salesDetail;
```

Copyright 2012 Asakusa Framework Team All Rights Reserved.

```
    In<ItemInfo> itemInfo;
    Out<JoinedSalesInfo> categoryA;
    Out<JoinedSalesInfo> categoryB;
    protected void describe() {
        // 商品マスタと売上明細を結合する
        JoinItemInfo joinItemInfo = operator.joinItemInfo(itemInfo, salesDetail);
        // カテゴリーコードで分岐する
        Separate separated = operator.separate(joinItemInfo);
        // フローの出力にそれぞれ結線する
        categoryA.add(separated.categoryA);
        categoryB.add(separated.categoryB);
        // 結合に失敗した場合は停止する（出力しない）
        core.stop(joinItemInfo.missed);
        // 分岐条件に合致しなかったデータを停止する（出力しない）
        core.stop(separated.unknown);
    }
```

データの結合はせずに、商品マスタと売上明細を商品コードで突き合わせて、カテゴリコード別に出力を振り分ける処理は、MasterBranch を利用して以下のようになる。

[Operator DSL]

```
    public enum ResultType {
        CATEGORY_A,
        CATEGORY_B,
        UNKNOWN;
    }

    @MasterBranch
    public ResultType separate(
            @Key(group = "itemCode") ItemInfo itemInfo,
            @Key(group = "itemCode") SalesDetail) {
        if (itemInfo == null) {
            // マスタが存在しない場合は null になる。
            return UNKNOWN;
        }
        if (itemInfo.getCategoryCode().equals("01")) {
            return CATEGORY_A;
        } else if (itemInfo.getCategoryCode().equals("02")) {
            return CATEGORY_B;
        } else {
            return UNKNOWN;
        }
    }
```

[Flow DSL]

```
    In<SalesDetail> salesDetail;
    In<ItemInfo> itemInfo;
    Out<SalesDetail> categoryA;
    Out<SalesDetail> categoryB;
    protected void describe() {
        // 商品マスタと売上明細を突き合わせてカテゴリーコードで分岐する
        Separate separated = operator.separate(itemInfo, salesDetail);
        // フローの出力にそれぞれ結線する
        categoryA.add(separated.categoryA);
        categoryB.add(separated.categoryB);
        // 分岐条件に合致しなかったデータを停止する（出力しない）
        core.stop(separated.unknown);
    }
```

Copyright 2012 Asakusa Framework Team All Rights Reserved.

4.2.3. 多数の種類のマスターや Tx と複雑な結合処理を行う場合

この場合は、基本的に演算子として CoGroup を選択する。これは任意の入力をとり、結合処理が可能で、かつ、任意の出力ができるというほぼオールマイティーな演算子である。CoGroup の制限的な利用が上記の存在チェックやマスター結合になっているとも言える。上記結合処理は CoGroup で記述することは可能であるが、これは**推奨されない。CoGroup 自体は最適化がかかりにくい実装になっている。従って CoGroup しか手段がない場合にのみ利用すべきである。**

以下のような結合処理の具体例を提示する。
商品コードをキーに受注明細、出荷明細、在庫マスタを結合する。
結合条件は、受注の赤黒伝票で訂正後の受注明細の受注 No で出荷明細と結合する。
商品コードで在庫の利用可能数が出荷予定数を超えていないことを確認する。
超えていた場合は、出荷エラーデータを出力する。
引当ができればピッキングデータを出力する。
引当済数と利用可能数を更新した在庫データを出力する。

[DMDL]

```
"受注"
order = {
    "受注 No"
    order_number : TEXT;
    item_code : TEXT;
    order_quantity : INT;
    unit_price : INT;
    "訂正伝票番号"
    correct_number : TEXT;
};
"出荷"
shipment = order + {
    shipment_number : TEXT;
    ship_to_code : TEXT;
};
"在庫"
inventory = {
    item_code : TEXT;
    "現在庫"
    actual : INT;
    "引当済在庫"
    reserved : INT;
    "利用可能在庫"
    available : INT;
};
"ピッキング"
picking_list = shipment + {
    shipment_status : TEXT;
};
```

[Operator DSL]

```
public abstract class PickingOperator {
    private PickingList cache = new PickingList();
    @CoGroup(inputBuffer = InputBuffer.ESCAPE)
    public void createShipmentAdvice(
```

Copyright 2012 Asakusa Framework Team All Rights Reserved.

```
            @Key(group = {"item_code"}, order = "order_number") List<Order> orders,
            @Key(group = {"item_code"}, order = "order_number") List<Shipment> shipments,
            @Key(group = {"item_code"}) List<Inventory> inventories,
            Result<PickingList> picking,
            Result<Inventory> availableInventory,
            Result<Order> error) {
        // Inventory は、item_code でユニークなので、１件のみ。
        Inventory inventory = inventories.get(0);
        for (Order order : orders) {
            if (!order.getCorrectNumberOption().isNull()) {
                // 訂正伝票は処理対象外とする。
                continue;
            }
            for (Shipment shipment : shipments) {
                if (order.getOrderNumber().equals(shipment.getOrderNumber())) {
                    int reserved = inventory.getReserved() + order.getOrderQuantity();
                    int available = inventory.getActual() - reserved;
                    if (available < 0) {
                        // 引当できなければエラーとする。
                        error.add(order);
                        break;
                    }
                    cache.setOrderNumber(order.getOrderNumber());
                    cache.setItemCode(order.getItemCode());
                    cache.setUnitPrice(order.getUnitPrice());
                    cache.setShipToCode(shipment.getShipToCode());
                    picking.add(cache);
                    // オブジェクトを再利用するため、出力後に null 値で初期化する。
                    cache.reset();[2]
                    inventory.setReserved(reserved); // 引当済数
                    inventory.setAvailable(available); // 利用可能数
                }
            }
        }
        availableInventory.add(inventory);
    }
}
```

4.3. *集計処理の場合*

Create/Update の処理で、その処理が集計処理の場合は DMDL で直接処理を記述する方法と演算子による方法がある。

単純集計であれば、DMDL で記述することで、演算子実装を省くことができる。集計モデルで定義できる集計関数[3]で要件を満たせるのであれば、DMDL で記述することも検討されても良い。この場合の演算子注釈は Summarize を利用する。

[2] DMDL から自動生成されたモデルクラスのメソッド。Null 値でプロパティを初期化する。
[3] http://asakusafw.s3.amazonaws.com/documents/0.2/develop/ja/html/dmdl/user-guide.html#id11

Copyright 2012 Asakusa Framework Team All Rights Reserved.

また、集計処理も結合処理と同じく、普通の演算子で記述可能である。複雑な条件による集計処理を行う場合はこちらを選択する。演算子は Fold を利用する。Fold は多重集計を行うようなケースにも利用することができる。

[単純集計の例]
・DMDL を使った単純なフィールドの合計値を計算するような場合の例

店舗コードと商品コード別に数量と金額を集計する。
[DMDL]
```
"売上明細"
sales_detail = {
    shop_code : TEXT;
    item_code : TEXT;
    quantity : INT;
    price : INT;
};

"商品コード別売上集計"
summarized sales_summary = sales_detail => {
    any shop_code -> shop_code;
    any item_code -> item_code;
    sum quantity -> quantity;
    sum price -> price;
} % shop_code, item_code;
```

[Operator DSL]
売上明細(SalesDetail)の集計結果として集計モデルの SalesSummary に変換される。

```
    @Summarize
    public abstract SalesSummary summarizeSales(SalesDetail salesDetail);
```

[Flow DSL]
```
    In<SalesDetail> salesDetail;
    Out<SalesSummary> out;
    protected void describe() {
        SummarizeSales summarized = operator.summarizeSales(salesDetail);
        // 処理結果の型は、集計モデルに変換されている。
        out.add(summarized.out);
    }
```

[複雑な条件の集計]
・以下のような複雑な条件で集計処理を行う場合は Fold を利用する。

売上数量と売上金額を店舗コード・商品コード別に集計する。
また、売上番号の重複を削除してカウント(客数)を集計する。

[DMDL]
```
"売上明細"
sales_detail = {
    "売上番号"
    sales_number : INT;
    shop_code : TEXT;
    item_code : TEXT;
    quantity : INT;
    price : INT;
    "客数"
```

Copyright 2012 Asakusa Framework Team All Rights Reserved.

```
    customers : INT;
};
```

[Operator DSL]

```
    @Fold
    public void summarizeSales(
            @Key(group = {"shop_code", "item_code"}, order = "salesNumber") SalesDetail sum,
            SalesDetail each) {
        if (!sum.getSalesNumberOption().equals(each.getSalesNumberOption())) {
            sum.setCustomers(sum.getCustomers() + 1);
            sum.setSalesNumber(each.getSalesNumber());
        }
        sum.setQuantity(sum.getQuantity() + each.getQuantity());
        sum.setPrice(sum.getPrice() + each.getPrice());
    }
```

[Flow DSL]

```
    In<SalesDetail> salesDetail;
    Out<SalesDetail> out;
    protected void describe() {
        SummarizeSales summarized = operator.summarizeSales(salesDetail);
        out.add(summarized.out);
    }
```

[Fold を利用した多重集計の例]

組み上げ計算、ツリー構造トレース、多次元集計等の複数軸での集計処理を行う場合は,
結合キーをフラットデータに組み上げ直して、並列分散集計を行う手法が、「現時点」では効率が良い。具体的な実装は以下になる。以下の例は 3 階層構造での集計のサンプルの例である。尚、ツリー構造をフラットにする処理は省略している。

[DMDL]

```
"売上情報"
sales_info = {
    "商品分類(大)"
    major_category_code : INT;
    "商品分類(小)"
    minor_category_code : INT;
    "商品コード"
    item_code : TEXT;
    "販売数量"
    quantity : INT;
    "販売金額"
    amount : INT;
};
```

[Operator DSL]

```
    /**
     * 商品分類（大）の集計
     */
    @Fold
    public void summarizeMajorCategory(
            @Key(group = "major_category_code") SalesInfo sum, SalesInfo each) {
        addValues(sum, each);
    }
    /**
     * 商品分類（小）の集計
     */
```

Copyright 2012 Asakusa Framework Team All Rights Reserved.

```
    @Fold
    publi void summarizeMinorCategory(
            @Key(group = "minor_category_code") SalesInfo sum, SalesInfo each) {
        addValues(sum, each);
    }
    /**
     * 単品の集計
     */
    @Fold
    publi void summarizeItem(
            @Key(group = "item_code") SalesInfo sum, SalesInfo each) {
        addValues(sum, each);
    }

    private void addValues(SelasInfo sum, SalesInfo each) {
        sum.setQuantity(sum.getQuantity() + each.getQunatity());
        sum.setAmount(sum.getAmount() + each.getAmount());
    }
```

```
[ Flow DSL ]
    In<SalesInfo> salesInfo;
    Out<SalesInfo> majorCategorySummary;
    Out<SalesInfo> minorCategorySummary;
    Out<SalesInfo> itemSummary;
    protected void describe() {
        // 商品分類（大）の集計
        SummarizeMajorCategory summarizedMajor = op.summarizeMajorCategory(salesInfo);
        majorCategorySummary.add(summarizedMajor.out);
        // 商品分類（小）の集計
        SummarizeMinorCategory summarizedMinor = op.summarizeMinorCategory(salesInfo);
        minorCategorySummary.add(summarizedMinor.out);
        // 単品の集計
        SummarizeItem summarizedItem = op.summarizeItem(salesInfo);
        itemSummary.add(summarizedItem.out);
    }
```

4.4. *グルーピング処理*

Create/Update の処理で、一定のキーでデータの集合を作るような場合は GroupSort 演算子を利用する。いわゆる GroupBy と同じ処理になる。様々な応用が可能である。

[GroupBy で集計処理のケース]
取引先データを集計して、伝票イメージを作成し、6 行ごとに改ページして、合計金額をセットするようなケースは以下のように記述する。

```
[ DMDL ]
"売上情報"
customer_sales_info = {
    "取引先コード"
    customer_code : INT;
    "商品コード"
    jan_code : LONG;
    "販売金額"
    amount : INT;
};
"売上伝票"
sales_slip = {
    customer_code : INT;
```

Copyright 2012 Asakusa Framework Team All Rights Reserved.

```
    detail_line_1 : TEXT;
    detail_line_2 : TEXT;
    detail_line_3 : TEXT;
    detail_line_4 : TEXT;
    detail_line_5 : TEXT;
    detail_line_6 : TEXT;
    total_amount : INT;
    page_number : INT;
};
```

[Operator DSL]

```
    @GroupSort
    public void createSalesSlip(
            @Key(group = "customer_code", order = "jan_code") List<CustomerSalesInfo> list,
            Result<SalesSlip> out) {
        Iterator<CustomerSalesInfo> itr = list.iterator();
        for (int page = 1; itr.hasNext(); page++) {
            CustomerSalesInfo info = itr.next();
            int totalAmount = info.getAmount();
            SalesSlip slip = new SalesSlip();
            slip.setCustomerCode(info.getCustomerCode());
            slip.setDetailLine1AsString(formatLine(info));
            if (itr.hasNext()) {
                info = itr.next();
                totalAmount += info.getAmount();
                slip.setDetailLine2AsString(formatLine(info));
            }
            if (itr.hasNext()) {
                info = itr.next();
                totalAmount += info.getAmount();
                slip.setDetailLine3AsString(formatLine(info));
            }
            if (itr.hasNext()) {
                info = itr.next();
                totalAmount += info.getAmount();
                slip.setDetailLine4AsString(formatLine(info));
            }
            if (itr.hasNext()) {
                info = itr.next();
                totalAmount += info.getAmount();
                slip.setDetailLine5AsString(formatLine(info));
            }
            if (itr.hasNext()) {
                info = itr.next();
                totalAmount += info.getAmount();
                slip.setDetailLine6AsString(formatLine(info));
            }
            slip.setTotalAmount(totalAmount);
            slip.setPageNumber(page);
            out.add(slip);
        }
    }
    private String formatLine(CustomerSalesInfo info) {
        return String.format("%d\t%d", info.getJanCode(), info.getAmount());
    }
```

```
[ Flow DSL ]
    In<CustomerSalesInfo> salesInfo;
    Out<SalesSlip> salesSlip;
    protected void describe() {
        // 取引先別、売上伝票作成
        CreateSalesSlip slip = operator.createSalesSlip(salesInfo);
```

Copyright 2012 Asakusa Framework Team All Rights Reserved.

```
        salesSlip.add(slip.out);
    }
```

4.5. *フィールドの変形処理*

Create の特殊なケースで、前処理や後処理で特有の処理になるときに利用する。各フィールドをばらす、またはまとめる処理を行うケースで利用する。

データ・フィールドをバラすような処理の時に利用する。実際の使用例は以下のとおり、通常価格と特別価格でデータを分割するケース。

[DMDL]

```
"出荷伝票"
shipment = {
    shipment_code : TEXT;
    buyer_code : TEXT;
    item_code : TEXT;
    units : INT;
    selling_price : INT;
    special_selling_price : INT;
};
"出荷明細（通常価格）"
normal_price_detail = {
    item_code : TEXT;
    units : INT;
    selling_price : INT;
};
 "出荷明細（特別価格）"
special_price_detail = {
    item_code : TEXT;
    units : INT;
    special_selling_price : INT;
};
```

[Operator DSL]

```
    private NormalPriceDetail normal = new NormalPriceDetail();
    private SpecialPriceDetail special = new SpecialPriceDetail();
    @Extract
    public void extractDetails(Shipment shipment,
            Result<NormalPriceDetail> normalOut,
            Result<SpecialPriceDetail> specialOut) {
        normal.setItemCode(shipment.getItemCode());
        normal.setUnits(shipment.getUnits());
        normal.setSellingPrice(shipment.getSellingPrice());
        normalOut.add(normal);
        // 特別価格が設定されていれば、特別価格を出力する。
        if (!shipment.getSpecialSellingPriceOption().isNull());
            special.setItemCode(shipment.getItemCode());
            special.setUnits(shipment.getUnits());
            special.setSpecialSellingPrice(shipment.getSpecialSellingPrice());
            specialOut.add(special);
        }
```

Copyright 2012 Asakusa Framework Team All Rights Reserved.

```
    }
```

5. フロー制御の割当

データ･フィールドに対する CRUD ではなく、データフローの分岐等が発生している場合に対しては、適切なフロー制御の演算子を割り当てる。データモデルが分岐するので、分岐後のフローを独立させて設計上は記述する。それぞれの具体的なフロー制御は以下の通り。

[条件をつけて分岐処理をさせる時]
条件をつけてデータ処理を分岐させるようなケースに利用する。演算子の内部でマッチングをおこなって、分岐処理をする。Branch を利用する。フラグをチェックして処理を分岐させるような場合は以下のようになる。

[DMDL]

```
"出荷伝票"
shipment = {
    shipment_code : TEXT;
    units : INT;
    selling_price : INT;
    "納品区分"
    delivery_type : TEXT;
};
```

[Operator DSL]

```
    public enum ResultType {
        DIRECT,
        CENTER,
        UNKNOWN;
    }

    @Branch
    public ResultType separate(Shipment shipment) {
        if (shipment.getDeliveryType().equals("01")) {
            return DIRECT;
        } else if (shipment.getDeliveryType().equals("02")) {
            return CENTER;
        } else {
            return UNKNOWN;
        }
    }
```

[Flow DSL]

```
    In<Shipment> shipment;
    Out<Shipment> direct;
    Out<Shipment> center;
    protected void describe() {
        // 納品区分で分岐する
        Separate separated = operator.separate(shipment);
        // フローの出力にそれぞれ結線する
        direct.add(separated.direct);
        center.add(separated.center);
        // 分岐条件に合致しなかったデータを停止する（出力しない）
        core.stop(separated.unknown);
    }
```

Copyright 2012 Asakusa Framework Team All Rights Reserved.

[結合処理をして複数の出力先を指定する場合]
何らかの結合をおこなってその結果として分岐処理をさせる場合は、MasterBranch を利用することが近道になるだろう。

[DMDL]
```
"得意先マスタ"
buyer_info = {
    buyer_code : TEXT;
    name : TEXT;
    trade_type_code : TEXT;
};
"出荷伝票"
shipment = {
    shipment_code : TEXT;
    buyer_code : TEXT;
    units : INT;
    selling_price : INT;
};
```

[Operator DSL]
```
    public enum ResultType {
        INITIAL,
        EXISTING,
        UNKNOWN;
    }

    @MasterBranch
    public ResultType select(
            @Key(group = "buyer_code") BuyerInfo buyerInfo,
            @Key(group = "buyer_code") Shipment shipment) {
        if (buyerInfo == null) {
            return UNKNOWN;
        }
        if (buyerInfo.getTradeTypeCode().equals("01")) {
            return INITIAL;
        } else if (buyerInfo.getTradeTypeCode().equals("02")) {
            return EXISTING;
        } else {
            return UNKNOWN;
        }
    }
```

[Flow DSL]
```
    In<BuyerInfo> buyerInfo;
    In<Shipment> shipment;
    Out<Shipment> initial;
    Out<Shipment> existing;
    protected void describe() {
        // 納品区分で分岐する
        Separate separated = operator.separate(shipment);
        // フローの出力にそれぞれ結線する
        initial.add(separated.initial);
        existing.add(separated.existing);
        // 分岐条件に合致しなかったデータを停止する（出力しない）
```

```
        core.stop(separated.unknown);
    }
```

[データとデータを一緒にする(UNION)場合]
ばらばらに処理をした伝票を束ねるような同じデータタイプを合流させるような合流処理を行う場合には Confluent を利用する。同じデータ・フィールドの種別で分流してそれぞれの処理を行い、その後で合流させるようなケースで利用する。エラー処理を行って、正常系に復帰させる時にも利用する。分流させることで並列処理が可能になるので、うまく Branch 処理と併用すると効率があがる。

[DMDL]

```
 "出荷伝票"
shipment = {
    shipment_code : TEXT;
    buyer_code : TEXT;
    units : INT;
    selling_price : INT;
};
```

[Flow DSL]

```
    In<Shipment> slip1, slip2, slip3;
    Out<Shipment> out;
    protected void describe() {
        CoreOperatorFactory core = new CoreOperatorFactory();
        Confluent<Shipment> operator = core.confluent(slip1, slip2, slip3);
        out.add(operator);
    }
```

Copyright 2012 Asakusa Framework Team All Rights Reserved.