Asakusa DSL 設計手法

改訂履歴

版	日付	内容
1.0	2012/01/17	新規作成

1. はじめに

AsakusaDSL での設計手法を述べる。この設計手法は分散環境でのバッチ処理の設計を目的としている。前提として、Asakusa でのバッチ・ジョブフロー・フローパート・演算子の役割は理解しているものとする。この文書では、具体的にどのようにバッチ・ジョブフロー・フローパート・演算子を設計するかを述べる。

ここでの「設計」とは後段で説明するように、一般に「基本設計」といわれるものを意図している。実装を細かく規定する詳細設計は、この文書ではなく、詳細設計・実装技法で述べる。また、この設計手法はあくまで参考例であり、実際の適用については、プロジェクトの規模・内容・人員によって、それぞれ検討すること。

2. 基本設計の定義と位置付け

ここでは開発サイクルとして、要件定義->基本設計->詳細設計->実装->テスト->リリースの開発フローを想定する。その中で基本設計は、要件定義での成果物を入力とし、基本設計書を成果物として生成し、詳細設計に適切な情報を提供する。また、基本設計書をもとに、それに対応するテスト計画書が作成され、適切なバッチ処理のテストを実施する。

AsakusaDSL での基本設計においては、データ・フローとデータ・タイプを明確にする。データ・フローは、バッチ処理におけるデータの処理とそのつながりを表現し、データ・タイプはそのフローの入出力の型を表現する。両者を組み合わせて、データの Input/Process/Output を表現する。この Input/Process/Output を Asakusa のバッチ・ジョブフロー・フローパートで表現する。これが基本設計のターゲットになる。

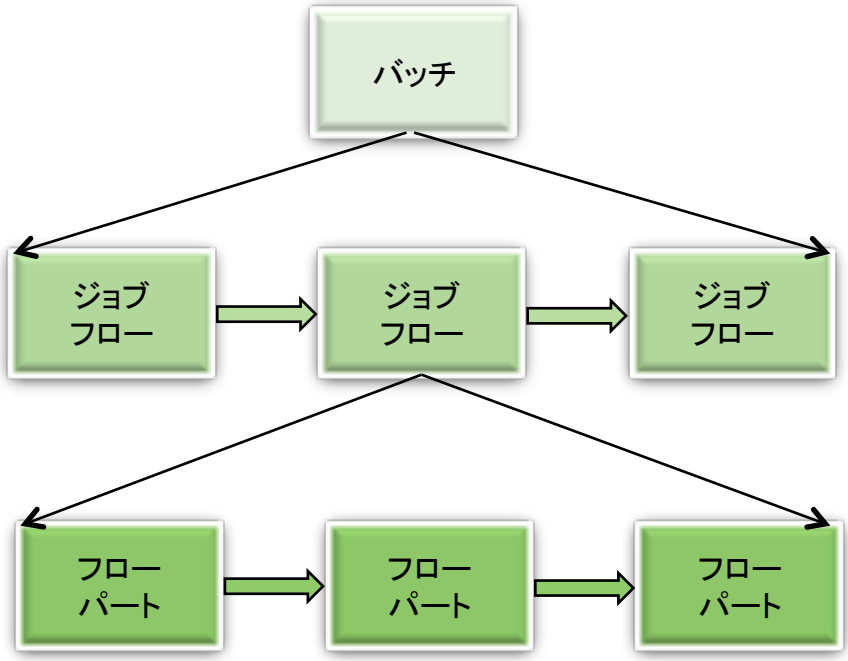

図 1

3. 基本設計書への入力

要件定義で作成されるアウトプット（例：要件定義書）が基本設計書への入力になる。要件定義のアウトプットとして、基本設計では「バッチ処理の入出力の業務フォーマットと業務的な概要」の記述が必要である。

バッチ処理の入出力の業務フォーマットでは、バッチ処理で処理されるべきデータの入力ファイル・フォーマットと、バッチ処理で生成されるべきデータの出力ファイル・フォーマットの記述と、それぞれの業務的なセマンティクスの記述が必要である。各入出力フォーマットは、ファイル名・含まれるフィールド名・タイプ等必要な情報とフィールドに対する適

切な説明を含む。バッチ処理の概要では、バッチ処理の目的、そのために行われている処理の概要の記述が必要である。

要件定義全体では、これ以外のさまざまな情報が必要になるが、ここでは触れない。
バッチ処理を設計する場合は、最低限、上記の情報のインプットが必要である。

4. 出力される基本設計書

基本設計書は、詳細設計への入力に利用されるバッチ処理の概要設計書になる。詳細設計を行うために、基本設計書はバッチ・ジョブフロー・フローパートの記述情報を含むことが必要である。バッチ処理の意図している内容を、第三者に伝わるように基本設計書に記述し、詳細設計につなげることが、基本設計の目的である。以下、まず典型的な基本設計書の構成要素を示し、つぎに基本設計の指針について述べる。

4.1. *バッチ一覧*

バッチの一覧は、個々のバッチの概要を示す。バッチは、業務的な単位であり、処理の最上位の単位である。複数または単数のジョブフローがバッチを構成し、複数または単数のフローパートが、ジョブフローを構成する。
作成されるバッチの一覧はバッチ名等管理情報を含む、各バッチの構成内容を記述する。バッチはエンド・ユーザーが理解できる単位での記述が望ましい。

4.2. *ジョブフロー一覧*

システムの一定の処理の固まりをジョブと呼ぶ。必要に応じてトランザクション処理される最大の単位になる。AsakusaDSL ではこのジョブをジョブフローと呼んでいる。再利用性や実装容易性の観点から、それぞれのジョブフローを後述するフローパートに分解することが望ましい。
ジョブフロー一覧は、バッチにおける複数のジョブフローについての一覧をさす。基本的にバッチ単位で作成される。また、ジョブフロー一覧では、ジョブフローごとの処理の管理情報・処理概要を記述し、ジョブフロー単位で利用される入出力ファイル・フォーマットの一覧を記述する。

4.3. *ジョブフロー設計書*

ジョブフロー一覧で表示されている個々のジョブフローの詳細記述を含む。後述する DAG で記述する。原則としてデータフローを記述し、どのような入力データに対して、どのような処理を行い、どのような出力を行うのかを記述する。論理的な構成は、いわゆる HIPO(Hierarchy Input Process Output)方式になる。個々のジョブフローの詳細記述は以下の項目を含む。

・名称・ID 等のジョブフロー管理項目

・インプットデータモデルとアウトプットデータモデル
これはジョブフローへのエントリーのデータモデルとジョブフローにより生成されるデータモデルである。

・処理フローの構成グラフ
後述するフローパートをつなぐグラフモデルをさす。

・ジョブフローを構成するフローパートの概要記述
ジョブフローを構成するフローパートの説明を含む。フローパートは入出力と処理から構成される。一つのフローパートは、再帰的に別のフローパートを形成することが可能である。フローパートを利用して、多層的にかつ構成的にジョブフローを形成することが、再利用性や可読性の観点から望ましい。フローパートは詳細設計で、単一の演算子になるか、または複数の演算子に展開される。基本設計の段階では、処理の概要と留意点を記述していれば事足りる。

5. 基本設計書の作成方針　データフロー

上記にあげた基本設計書の各項目は、以下の事項に留意しながら設計すること。

- DAG の作成を企図すること

バッチフローは基本的に DAG(Directed Acyclic Graph)=有向非循環グラフを利用して記述する。DAG は、頂点（Vertex）と辺（Edge）から構成されるグラフの中で、すべての辺に向きがあり、かつ順路が循環しない形式のものである。

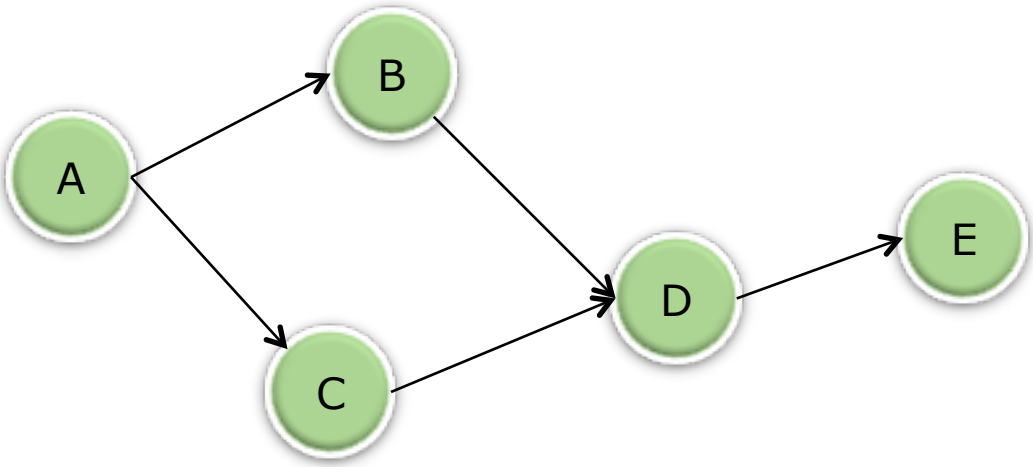

図 2

各頂点で処理を記述し、各辺でデータフローを記述することができる。バッチの出発地点に最初の頂点（図であればA）を設定し、以降バッチを構成する処理をそれぞれ隣接する頂点に記述し、その処理に渡すデータを辺で表す。

これは一般に構造化手法でいわれるところのデータフローダイアグラム（DFD）といわれるものに近い。ただし、DFDでは頂点にあたる部分にリソース（DB やファイルシステム）を記述するケースがあるため、厳密には DAG とは異なる。

一つの DAG ですべてを処理しようとすると、非常に複雑な表記になってしまうため、通常は後述するように階層構造を持たせて管理する。

5.1. *実際の DAG の書き方*

DAG のイメージは、前述の通りである。以下に実際に DAG を設計する際の指針を示す。

5.1.1. 「出口」を意識する

DAG には「入口」と「出口」がある。これがインプットデータとアウトプットデータにあたる。その「入口」と「出口」をつなぐ事が DAG の作成にあたる。まず DAG 作成の最初の出発点は「出口」がどこか？ということになる。すなわち、設計している処理では、どのようなアウトプットが欲しいのか？を明確に意識する事が最初に必要となる。そのアウトプットを作成するために必要なインプットデータは何か？と検討していくことにより「入口」にあたるインプットが明確になっていく。
すでにバッチ処理の要件定義で、インプットとアウトプットは事前に明示的に与えられていることが前提にはなっているが、まずはアウトプットに何が必要か？ということを明確に定義して置く事が肝要になる。

5.1.2. どのようにフィールドが形成されるか検討する

次に検討すべき事項は、アウトプットの各フィールドがどのように形成されているか？を検討していくことになる。必要なフィールドが、どのインプットに基づいて生成されるかを検討する。インプットがそのままコピーされるケースもあれば、インプットのフィールドを元に演算により生成されることもあるだろう。これにより必要なデータをインプットとして列挙していく。

5.1.3. 形成の過程を精査する

アウトプットのデータから、必要だと思われるインプットデータが列挙できた段階で、アウトプットのデータのフィールドの生成過程を精査していく。業務バッチにおいては、その生成過程は、データの検証とデータの操作に分かれることが多い。いかに述べる。

・　データの検証

値の検査になる。入力されたフィールドを検査して、適切かどうかテストする。検査の手法は、大きく形式的なものと意味的なものの二つに分かれる。形式的なものは、存在・型・範囲・桁数・文字数等の「形式により真偽」が明確なものの検査になる。意味的な検査は、「与えられたコンテキストでの真偽」の判定を行う検査になる。この数値は正しいのか？そもそも許されるのか？等の判定になる。検査されるデータそれ自体では判定はできないことが多いので、なんらかのデータの結合によって真偽を判定することが多い。一般にデータのクリーニングやクレンジングと言われる処理である。

・　データの操作

値の操作になる。入力されたフィールドを元に、データの読み出しや更新、値の生成や削除を行う。特に、値の更新や生成を処理として切り出していく。更新や生成は、コピー・分割・合流といった操作によるものや値の演算処理によることが多い。

これらの過程を、抽出しつつ DAG の頂点を浮き彫りにしていく。

5.1.4. 例外処理を考える

つぎに検討すべきは例外処理になる。これはフィールドの形成過程を検討しつつ同時に考慮していく。一度正常系を描いてから、異常系を検出していく手法でも問題はない。例外処理は通常は分岐処理になることが多く、特に業務系の例外処理は、分岐した正常処理として扱った方が適切である。この例外系の設計は、極めて重要であり決して軽視してはならない。特に実装以降での例外処理の手戻り設計は、システム全体の品質・運用・メンテナビリティに大きな影響を及ぼす。
基本的には、フィールドの形成過程で失敗した場合の処理を検討しつつ、DAG　の構成要素として検討していく。これについては後述する。

5.1.5. フローをくみ上げる

以上の検討を行いつつ、データフローを記述していく。この時に考慮すべきはデータの分岐処理である。フィールドの生成過程を分岐・合流等を利用しながら一つの流れに仕上げていく。

5.1.6. 順序を検査する

フローのくみ上げの過程で、「処理の順序性」を考慮していく必要がある。まずは依存関係による順序性の整理を行う。これは依存度の高いものほど「前に」＝付け足し的なものほど「後に」もっていく必要がある。次に重要度であるが、これは原則として致命的のものほど「前に」＝　軽微なものほど「後に」もっていくことが望ましい。「フェイル・ファーストの原則」は必ず意識すること。

5.2. *階層構造を意識する*

DAG を記述する時には、通常多層的に構成していく。DAG で構成されたジョブフローを下図のようにドリル・ダウンさせていく。

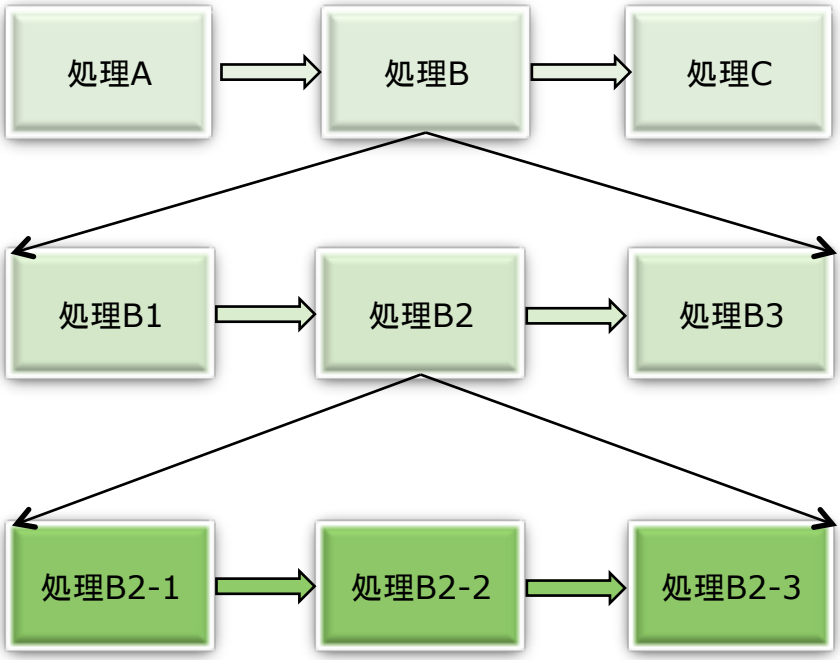

図 3

ジョブフローは一番詳細な処理を並べて記述するときわめて煩雑なフローになってしまう。多層性がなくフラットな一本線のフローになってしまう。また逆に処理をまとめすぎると、不用意に階層が深くなり一覧性に欠ける。適切な深さで、適切な粒度感でジョブフローを記述することが必要である。

階層が深くなり過ぎるとメンテナンス性が落ちるため、通常は 3-4 階層程度の階層の深さが望ましい。レイヤーごとにレベルを持たせて（すなわち、レベル 0、レベル 1、レベル2・・・）ある程度の役割を持たせることが必要になる。各レイヤーの目安は以下の通りになる。

レベル 0：最上の階層なので、一覧性を優先させること。
レベル 1：レベル 0 を受けて、一定の業務処理の固まりを分解したレイヤーであることがおおい。レベル 0 とレベル 1 で処理の概要は、一覧で理解できる程度に記述すること。
レベル 2：これ以降は基本設計の深さに依存する。一覧性よりも上位レベルでの処理の詳細な説明を行うことを優先する。

各処理の粒度感をどのように判断していくかは、熟慮を要する。基本的には、業務トランザクション単位で分割すること。この時、後述するロールバックポイントが、必ず処理の境界になる。さらなる処理の分割方法は後述する。

5.3. *ロールバックポイント・リカバリーポイントの認識*

DAG の辺（edge）には、トランザクション境界を設定することができる。なんらかの処理が途中で失敗した場合に、データフローのどの時点まで処理を戻すかを規定する。バッチ処理でのトランザクションであるので、オンライン処理のような非常に細かい単位でのトランザクションではない。ロング・トランザクションになる。

ロールバックポイントは、一義的には要件定義で作成される業務フローを参考にして設定する。一般に、業務フローは、業務の簡単な流れをデータフローと業務関連当事者を中心に記述している。データに対する業務関連当事者（データオーナー）が変更される境界を判断し、処理のトランザクション境界とすることができる。利害関係者がかわる状態でのデータは通常はなんらかの確定処理が終了していることが多く、後続するデータ処理中に障害が発生した場合、処理を巻き戻して再スタートを行う場合に出発点として適切であることがおおい。

上記のトランザクション境界をジョブフロー間の境界として設定する。

5.4. *並列性の確認*

DAG の設計時点で、処理の並列性を確認すること。
現行のバッチ処理、特に汎用機のような多重度での処理によりパフォーマンスを出すようなアーキテクチャを採用しているシステムでのバッチ処理は、ほぼすべてで分散処理が可能である。

基本設計時点でバッチ処理の並行処理が可能かどうか、この時点で確認することが可能である。この時点で並列性が確認できるのであれば、大抵のケースでは分散処理は可能であることがおおい。ただし、処理の詳細設計時に隠れた要因が検出され、並列性に影響が出る場合もあることに留意すること。尚、この基本設計時点で、並列性に疑義があるばあいは、その処理自体の並列性に課題がある可能性が高く、分散処理自体を再検討すべきである。

基本設計レベルでの並行性の確認は、その業務を手作業で行う場合に、複数の担当者で同時に行うことができるかどうか？という粗いレベルでの確認になる。

例えば、請求データを作成するバッチ処理を考えてみる。システムになんらかの障害が発生して、人手で処理を代行する場合のオペレーション、または、そもそもシステム化以前に人手で処理をしていた時期のオペレーションを想定する。請求書を作成する作業は、果たして各人で分担して処理することが可能であるかどうか？判断する。可能であると判断できれば、おそらく並列処理が可能である可能性が高い。

逆に、業務の流れを確認している時に、なんらかのデータを一斉に参照したり、更新したりするようなことが想定できる場合は並列性の確保は難しい。例えば、前述の請求書の作成の例であれば、ある書類に複数の担当者が同時に書き込みをしなければならないような状況がこれにあたる。これでは担当者の数を増やしても請求書の作成作業のスピードをあげることは難しい。

基本設計での並列性の確認は、以上のように粗いレベルでの確認になる。もちろん詳細設計時に精査を行うが、最上位のフローレベルで既に並列性に問題がある場合は、この様に基本設計の段階で確認することができる。

5.5. *例外処理*

例外処理は、システム例外と業務例外に分けることができる。システム例外はエンドユーザーレベルでは起きてはいけないという前提で設計し、システム障害対策としてハンドルする。業務例外は「基本的に通常起きることがある」という前提で処理を設計する。

業務例外処理の設計のポイントは、まずカテゴリーとして以下の3種類に分類することと、業務的なロールバックポイントの検討と正常復帰業務ルールの設計になる。

業務例外処理は以下の3種類にカテゴライズする

5.5.1. 致命的なストップエラー

処理自体を中止するエラーを指す。処理を続行する意味がないエラー。エラーデータのインスタンスのみをストップさせるか、ジョブフロー全体をストップさせるか判断する必要がある。エラーデータのインスタンスのみをストップさせる場合は、エラーデータは分流させて別処理をおこない、正当なデータは通常の処理を続行させる。

エラーデータのみをストップさせる場合は、データ・インスタンス単位でのリランを考慮する必要があるため、バッチ自体の設計から検討する必要がある。検討手法の一つとして、ジョブフローのデータ依存性を可能な限り排除して、実行時の外部コンテキストで対象データの切り分けが出来るようにする方法がある。

5.5.2. 警告エラー

処理は続行するエラーになる。業務セマンティクスとしては正当ではないが、全体の処理が優先されるようなケースで、かつその続行が可能な場合のエラー。業務ルール依存になる。処理フロー自体は正常系に織り込む必要がある。ガード付きのマッチング処理やパターン処理でハンドリングを行う設計を行う。

5.5.3. ヒューリスティックなエラー

エラーセマンティクスがランタイム時に決定されるようなエラーで、人為的な判断が本来必要とされるようなエラー。データとしては不整合が起きている可能性があるが、与えられている処理のセマンティクスでは完全になってしまっているようなケースになる。滅多に起こらない（というかシステムでは本来判断できない）。システム上の閉じた系ではエラーではないので処理は大抵は続行されて、警告処理されることが通常になる。そのシステムを越えたより広い系で見た場合には、不整合が起きている可能性が高いので、エンドユーザーレベルでの判断が必要なエラー。

業務的なロールバックと正常復帰処理については、基本的には人為的な判断がはいるため、エラーデータの人為的なハンドリングのあとで実行されるバッチ処理によりリカバリーされる。この処理をしっかり設計しておくかどうかで、運用時の業務負荷が非常に大きく変わる。異常系処理の正常系復帰設計の手法は、このガイドの範疇を越えるので述べない。

6. 基本設計書の作成方針　データモデル

6.1. *DDL での入出力モデルの設計*

業務トランザクションを境界とするデータの入出力を明文化する。特に記述方式について制約はないが、通常は一般的な DDL での設計になる。テーブルモデルを想定するのであれば、データベース設計おけるテーブル設計に近い。テーブル名・カラム名・カラムのデータ型、また可能であれば制約条件といったもので構成されていることが望ましい。

6.2. *フロー設計との関連について*

データモデルを検討にするにあたり、外部入出力だけではなく、内部のフローパートや演算子という構成モジュールとの連携についても注意が必要である。プロセス指向で考えていくと過剰にデータ項目が増え、最終的にはシステムのメンテナンスが困難になる。

分散並列処理を前提にした非同期バッチ処理のデータモデルの設計では、データモデルを“渡り歩く”メタモデルを想定することが必要になる。

これまでの同期的なバッチ処理では、基本的に静的なデータモデルを想定し、それが変更・変形・更新されて別のデータモデルに状態が変わるという処理を想定していた。具体的には、ACID トランザクションを前提として、静的なモデルの状態遷移、つまりメンバーの変更やクラスの生成・削除・変更を、当初のデータモデルからの変更として設計する手法である。この手法は広く一般に利用されており、様々な局面で有効であり、かつ実績も豊富だ。

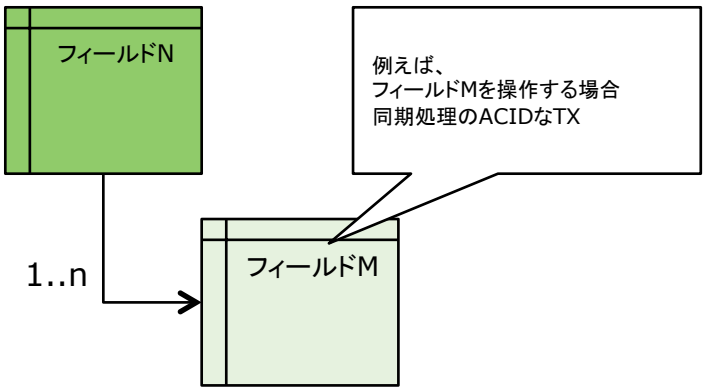

図 4

非同期バッチ処理でも同じように、単一の静的なモデルをベースにした設計手法を適用するという考え方はある。しかし並列処理を前提とした非同期バッチ処理では、そもそも前提となるトランザクションモデルが異なっているため、ベースにする実装モデルを別に想定した方が適切であることが多い。データモデルから別のデータモデルへ生成変形していくというデザインがより有効である。そうすることで分散並列処理が行いやすいモデルを獲得していくことが可能となる。

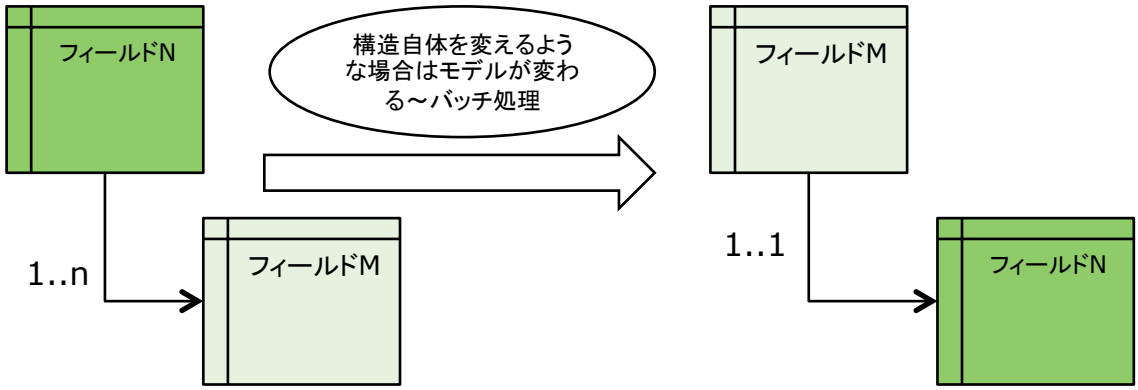

図 5

データモデルから別のデータモデルへ変換する処理が、まさに DAG の頂点に当たる。頂点では、複数のデータソースと複数のデータシンクが存在し、それぞれをデータモデルとして表現することが可能である。

他方、単一データモデルに比べて、上記の手法はデータモデルそれ自体が増加するために、データモデルが過剰にならないように注意する必要がある。

6.3. *マスターとトランザクションへの分離*

非同期バッチ処理上のデータモデルは、変形・生成の対象となるトランザクションデータと、その則を提供するマスターデータとに分けると良い。非同期バッチ処理では、マスターデータとトランザクションデータのジョイン（結合）などを工夫しないと効率良く並列実行できない処理が頻出する。その効率がシステム全体に大きく影響し、その設計を間違えると、およそ性能が出ない。

トランザクションデータとマスターデータの違いは一般に言われるほど自明ではない。一般にはマスターデータをもとにトランザクションデータが生成・変形・修正されることが多いが、トランザクションデータからマスターデータを生成することもある。予断をもってマスターデータとトランザクションデータを単純に分類してしまうと、後行程で大幅な修正が入ることがある。大元のエントリーポイントでの入力データと、最終出力の出力データは決まっており、それの処理に利用するデータがどのようなものかという観点で、トランザクションやマスターの切り分け設計を始めるべきである。

具体的にトランザクションデータとマスターデータに焦点を当ててデータモデルを作ることから検討する。この辺りは通常の正規化の手順と変わらない。

まず大事なのは一意性の保証、すなわち特に結合のためのキーの設計である。ただし、複数のデータモデルを結合しながら動作するというパラダイムである場合、キーの一意性はシステム的なものと業務的なものは切り分けて考える必要がある。システム的な一意性は常に確保される必要があるが、業務的な一意性はデータモデルが移り変わっていく過程で変化することが多い。処理に応じて、どのフィールドがキーになるのか、丁寧に見定める必要がある。尚、実装上は、システム的なキーと業務的なキーを組みあわて分散度を上げるという手法があるため、キーの設計はアプリケーションのセマンティクスを理解した上で、考えを整理しておくことが大切である。

尚、分散並列処理を前提とする場合は、キーの設計においては物理階層と論理階層をそれぞれ意識しておく事が必要になる。大抵の分散環境では、物理階層のキー（上記のシステム的なもの）によるデータ分散は黙示的に行われることが多いことに対して、論理階層のキー（上記の業務的なもの）についてのデータ分散は、アプリケーション・セマン

ティクスに依存するため、明示的に行う必要があることが（現在の技術では）必要であることが多い。両者を混同した設計ではデータが分散しないことがあるため、注意が必要である。

つぎに各データの振る舞いに注目してみる。一般にトランザクションデータは大量になり、マスターデータはそれほど大規模ではない。（勿論例外もある。）またトランザクションデータは頻繁に更新・生成されるが、マスターデータはそれほどではない。（これもまた例外がある。）トランザクションデータ自体は、相当に冗長性が高く、属性間で不整合が発生することがよくある。その修正にマスターデータを利用する。データモデルの変遷を分析していく過程で、そのような修正が発生すれば、そこでマスターデータとトランザクションデータを区別できる。ただし、上記はあくまで相対的なものである。あくまで、データフローを設計しながら、トランザクションとマスターを切り分けていくことが重要である。例えば、かなりの頻度で“マスターデータ”を更新し、その更新のために元データが“トランザクションデータ“である場合は、むしろ、“マスターデータ”がトランザクションデータであり、“トランザクションデータ“がマスターデータになる。

データのモデリングは、データのロック制御やデータ配布の方法にも影響する。細かい項目の精査よりも先に、各種のデータ内容と振る舞いを把握して、マスターデータとトランザクションデータの位置付けを見極めることが重要である。これが前述した、結合などの性能に影響していく。

■トランザクションデータのモデル化

分散並列処理ではトランザクションデータの設計のウェイトが非常に大きい。処理の分離レベルを確保することは特に並列処理のパフォーマンスをあげるためには必須である。トランザクションデータのモデルは、分離レベルに大きく影響する。経験的には以下のポイントを考慮すると良い。

・キーの冗長性
結合処理を行う場合のキーを冗長に持つ事を厭わないこと。結果的には、変更耐性に強いモデルになることが多い。過剰に持ちすぎるとメンテナビリティに影響するが、マスターデータほどに冗長性を排する必要はない。

・自己記述性
可能限り必要なデータ自分自身の内包するデータ構造にすること。分散処理に限らず、処理環境の想定水準が低い場合の非同期処理での実装設計の基本でもあるが、外部との結合処理のコストと内部でのデータ保持のコストでは、結合コストの方が圧倒的に高くつくことを考慮すること。

・参照制約を自明としない
キーの冗長性と自己記述性の強化は並列処理の高いパフォーマンスの源泉になるが、その代償として、参照制約の保証の程度が下がる事がある。したがって設計上は、参照制約保証を最初から是とするのではなく、適切なタイミングで、検査する処理または例外処理を想定すること。

■マスターデータのモデル化

マスターデータのモデル化については、他に参照とされるべき資料・書籍が多数に上るため、ここでは割愛する。また、現在の AsakusaDSL では ComplexDataModel をサポートしてないため、サポート後に記述する予定である。

尚、現在、特に分散環境上での非同期バッチ処理を中心にしたときの大規模なデータモデルの設計方法は、決定的なものが見つかっていない。上記の記述は Asakusa の実際のプロジェクトに基づくものであり、より適切な手法が見いだされることを期待する。

7. 基本設計書の作成方針　洗練化

データフローとデータモデルが一度組み上がった段階で洗練化をおこなっていく。DAG の形成としてはバッチ処理のビルディング・ブロックを作る設計になる。プロセスの細分化と、データモデルの細かな検討を進める過程で、モジュー

ル化についてのリファクタリングを行っていく。それがジョブフローのフローパートや演算子の設計の一助となり、再利用性が向上することにもなる。

フローパートや演算子のカプセル化を進めると、品質が向上するだけではなく、バッチ処理自身の再利用性の向上にもつながる。特定のフローパートや演算子を取り替えて、バッチ処理全体を変化に対応させられるため、仕様変更に対する耐久性が高くなる。内部の改良による保守性の向上も期待できる。挙動を変えずに将来の拡張に備えることも可能であり、さらなるリファクタリングが容易になる。

7.1. *フローパートへの分割方法*

ジョブフローをその構成部品である各フローパートに分解していく。このフローパートはさらに下位のフローパートに分解できる再帰的な構成になる。フローパート分解により、フローパートの再利用性があがり、ジョブフロー全体の見通しが良くなる。またテストデータが作成しやすくなり、全体的に品質が向上する。一般に構造の分解は7つ程度のブロックにわけていくと見やすいと言われている。特に根拠はないと思われるが経験則として考慮する。

このフローパートへの分解は概ね次の方法で行う。

7.2. *STS 分割法*

ジョブフローの中で STS のパターンになっているものを検出する。STS とは Source-Transfer-Sink の構成であり、データのインプット->変形->アウトプットの形になっているフローである。

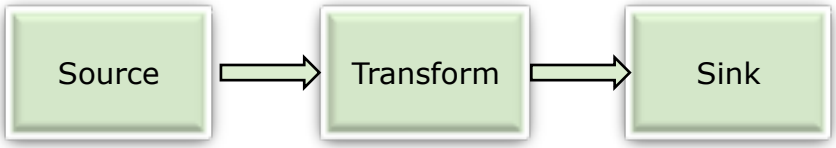

図 6

バッチ処理は通常は、事前データの処理、本処理、生成データの後処理の三つから構成されていることが多く、したがってバッチやジョブフローは、このような処理に分解することが可能だ。

前処理では、たいていは本処理に入るためにデータの整形や若干のデータの付加が行われる。後に続く本処理を実行するために、データを変形・チェック・生成する処理であることが多く、場合によっては簡単なクリーニングも行う。

それに続く本処理は、処理ロジックそのものにウェイトを置く。単純な処理の逐次実行にできれば見通しが良い。理想を言えば、本処理は完全に関数的なロジックであることが望ましい。それぞれの処理は入力に対して出力を行い、出力結果はグローバルな状態や前後の入力などに依存しない。副作用のあるような処理は前処理と後処理に置き、可能な限り純粋なロジックだけを本処理を実装する。これはテストの容易性の向上にもつながる。バグの混入が低減し、再利用性を高めることが可能になる。

例えば、Excel でのワークシートに組み込まれたセルの関数を思い浮かべてほしい。セルの関数自体はセルの値には依存せず、関数の変更はワークシートやセル自体に影響しない。このような関数を連続して呼び出すフローに設計しておく。これは、分散並列性を高めるということにもなる。

後処理は、データのアウトプットへの変形を行うことが多い。特にフォーマッティングや、データを複数の流れにコピーする処理、または、複数の流れからデータの合流をする処理を行う。最終的なアウトプットを生成するか、または別の次のフローにデータを受け渡す。

この STS 分割は再帰的に適用することができる。特に大規模なバッチの処理の場合は、多層構造に分解することができる。前処理、本処理、後処理のそれぞれ処理が下位の前処理、本処理、後処理に分かれ、さらに再帰的な分割を行うことができる。

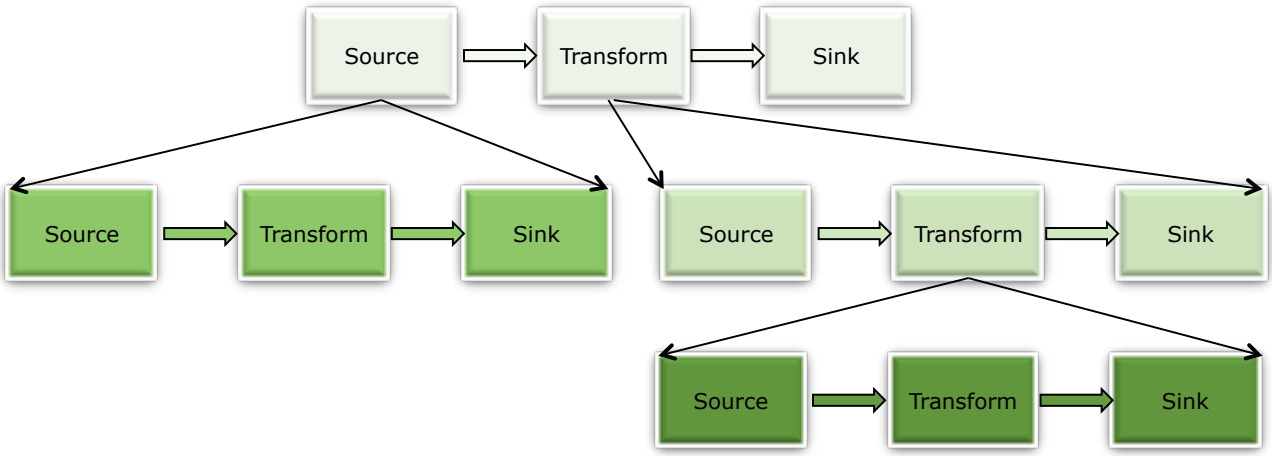

図 7

7.3. *凝縮性と関連性基準*

STS 分割を行ったあとで、課題になるのは Transform モジュールをどの程度に分割すべきなのか？または統合すべきなのか？ということになる。

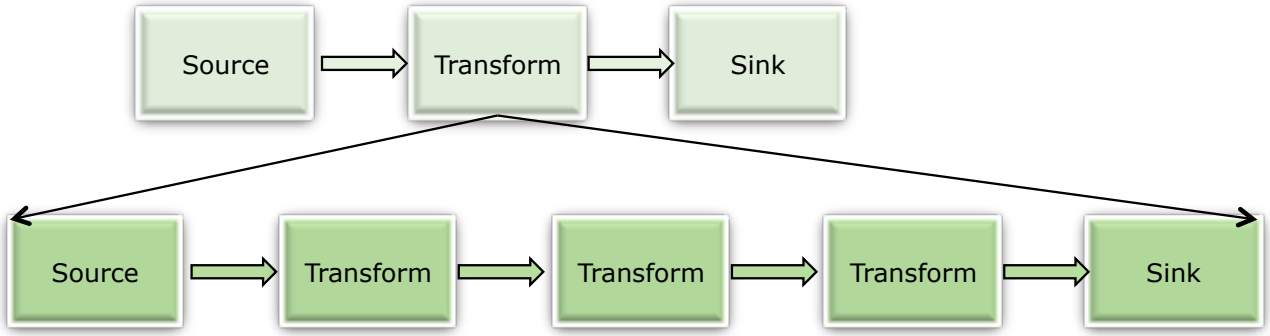

図 8

ここでの一つの基準として、粒度感の基準は、オブジェクト指向の GRASP パターンの核となる概念である、凝集度（Cohesion）と結合度（Coupling）を参考にする。それぞれの処理モジュールの、隣り合う処理モジュールとの依存性の高さ・低さを基準に考える考え方である。モジュール間で大量のデータの受け渡しをするのであれば、凝縮性は高く、一つの一方、受け渡しをするデータがそれほど多くなければ、関連性は低くモジュールとしての切れ目になる。オブジェクト指向の優れた設計は、「凝集度が高く、結合度が低い」と表現されることが多い。まさにこの基準でモジュールを分割していくと見通しがよい。

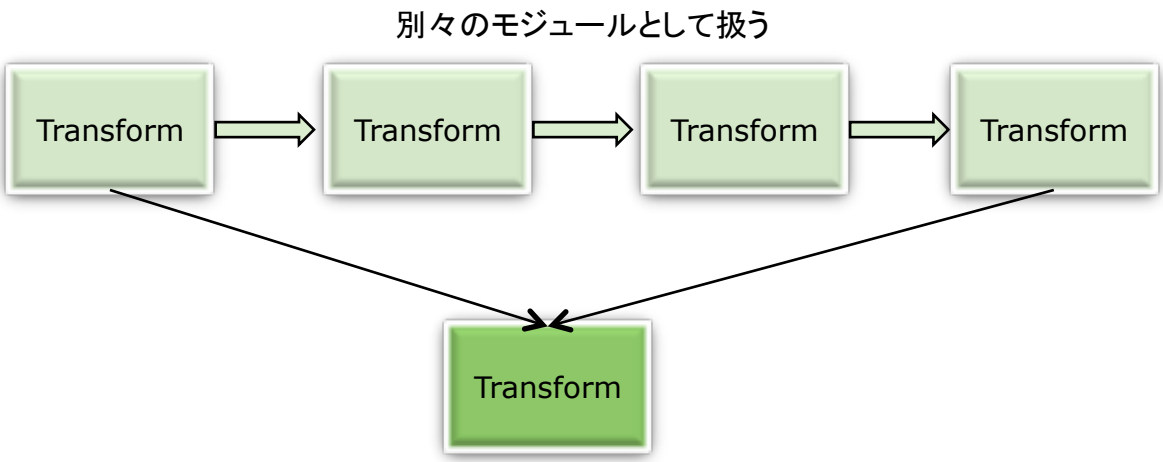

図 9

凝縮性・結合性のテストは、現実には、実際のケースで都度判断していく。この辺りは、構造化手法のモジュール強度の考え方も参考にできる。ほかにも、さまざまな考え方がある。

例としては、

①変更可能性（可換性）
②可読性
③業務的なセマンティクスの構成としての凝縮度

を基準にしていることも多い。モジュールの変更可能性（可換性）は重要である。業務フローの変更や、適用するプロダクトラインの変更、モジュールの類似する他の業務への流用が可能かどうかなどが判断基準にできる。