Javaバッチ実用化に向けたフレームワークの開発

野村総合研究所
開発技術部　上級テクニカルエンジニア

木村 綾太郎（きむら りょうたろう）

情報技術本部にてオブジェクト指向技術を活用したシステム開発に取り組むITエンジニア。金融、産業、流通など様々な業界のシステム向けにミドルウエアフレームワークを開発。これらの経験を生かして現在フレームワーク製品の開発業務に従事。

要旨

現状では、オンラインアプリケーションをJava言語で開発していても、バッチアプリケーションは従来通りCOBOL言語で開発するという事例が多い。この主な理由は、Javaバッチ（Java言語によるバッチ処理）に対する性能面での不安が大きいためであると考えられる。一方で、生産性向上へのさらなる要求、COBOL技術者確保への不安、Java実行環境の性能向上等を受けてJavaバッチへの期待は高まりつつある。

このような状況を受けて、筆者らは、Javaバッチの実用化に向けた取り組みを行った。まず、Javaバッチの性能向上を目指したRI（Record Item）フレームワークを開発し、Javaバッチの性能をCOBOLバッチと同等以上にまで引き上げられることを確認した。また、設計モデルにDFDを採用し、DFDの段階的詳細化により設計を行う手法により生産性向上を目指したFCI（Flow Control Injection）フレームワークを開発した。以上の取り組みにより、Javaバッチ実用化に対する目処が立ったと考える。

キーワード：Java、COBOL、バッチ、2007年問題、フレームワーク、DFD、ディレードオンライン、センターカット

Currently there are a lot of cases which even while online application is developed by Java language, batch application is still developed by COBOL language.The main cause should be uncertainty of Java batch performance, which is batch process using Java language. On the other hand, due to higher demand for productivity improvement, uncertainty of securing COBOL engineers, following performance improvement of Java runtime environment, expectation of Java batch is growing.

Under this situation, we took an approach toward the practical use of Java batch. First of all, we developed RI (Record Item) framework which worked toward performance improvement, confirmed to be able to improve performance of Java batch same as or more than COBOL batch.And we developed FCI (Flow Control Injection) framework which worked toward performance improvement through bringing in DFD as design model and design method using stepwise refinement of DFD. This approach shows the practical use of Java batch has good prospects.

Keyword : Java, COBOL, Batch, Year 2007 problem, Framework, DFD, Delayed online, Center cut

1．はじめに

現在、大規模システム構築においてバッチ処理をJava言語で開発するという事例はまだまだ少ない（本稿では、バッチ処理を、「大量のデータに対して一括処理を行う処理方式」の意味で用いる）。一方で、COBOL技術者確保への不安などを背景として、Javaバッチ（Java言語によるバッチ処理）に対する期待は高まりつつある。

このような状況を受けて、筆者らは、Javaバッチ実用化へ向けた以下のような取り組みを行った。

- Javaバッチの性能向上への取り組み
- Javaバッチの生産性向上への取り組み
- 様々なJavaバッチ処理方式への対応

本稿では、これらの取り組みについて報告する。

2．バッチ開発の現状とJavaバッチへの期待の高まり

（1）バッチ開発の現状

現在、Webアプリケーションが一般化したことにより、多くのプロジェクトがオンラインアプリケーションをJava言語で構築するようになった。一方で、このようなプロジェクトであってもバッチアプリケーションは従来通りCOBOL言語で構築している場合が多い。本来であれば、オンラインアプリケーションとバッチアプリケーションを共にCOBOL言語で開発していた時代のように、開発言語をJava言語に統一することが望まれるところである。

あえて、バッチアプリケーションを従来通りCOBOL言語で構築する主な理由は、以下の2つであると考えられる。

- Javaバッチに対する性能面の不安
  インタープリタ型の言語であるためJavaの実行速度は非常に低いという先入観から、Javaのバッチ処理への適用は敬遠されてきた。そのため、実際のバッチアプリケーションを使っての性能測定や性能向上のための工夫が行われて来なかった。この結果、Javaによるバッチ処理は遅いという先入観は拭われず、その適用が見送りになる状況が続いて来た。

- COBOLの生産性で十分という認識
  COBOL言語が利用されてきた歴史は古く、多くのリソースやノウハウの蓄積がある。このため、COBOL言語によるバッチ開発の生産性は高い。このような状況で、あえて未知のJava言語によるバッチ開発にチャレンジしようとするプロジェクトはほとんどなかった。

（2）Javaバッチへの期待の高まり

筆者らは、以下に示すようなシステム開発

を取り巻く様々な環境の変化を受けて、Javaバッチに対する期待は高まっていると考える。

●生産性向上へのさらなる要求
システム構築における短納期、低コスト開発に対する要求はますます高まりつつあり、これまでの生産性を「飛躍的」に向上させる必要がある。

●COBOL技術者確保への不安
一般に「2007年問題」と言われている団塊世代の大量退職が始まることで、これまでCOBOL開発を支えてきた技術者の確保が困難となることが予想される。また、一般化しつつあるオフショア開発においても、Java技術者に比べてCOBOL技術者の確保が難しいという状況がある。

●Java実行環境の性能／信頼性向上
Java実行環境に対する性能向上への取り組みは、継続的に行われている。例えば、インタープリタ実行の欠点を克服するためのHotSpotVM技術の登場により、Javaの実行速度は以前に比べて大幅に向上している。また、Webアプリケーションサーバでの稼動実績が蓄積されてきたことにより、安定性・信頼性ともに向上してきている。

●新たな開発技術の登場
AOP（アスペクト指向プログラミング）ツール、テスト自動化ツール等の新しい開発技術は、Java開発を取り巻くオープンソースプロジェクトから生み出されている。Java言語により開発を行っておけば、これらの開発技術を速やかに取り込み、生産性の向上に結びつけることができる。

3．Javaバッチの性能向上への取り組み

（1）性能向上への取り組みの概要

Javaバッチの性能向上を目指して以下のような取り組みを行った。

①現状のJavaバッチ性能の把握
COBOLバッチを比較対象とし、性能向上のための工夫を行わない状態でJavaバッチの性能を測定した。Javaバッチは、COBOLバッチと同様にファイル受け渡しのアーキテクチャで実装した。性能測定の結果、Javaバッチの性能は低いが工夫によりCOBOLバッチに追い付けるレベルであることがわかった。

②性能向上のための実装アーキテクチャの選定
処理間のデータ受け渡しの実装方式として、ファイル方式、RDB方式、メモリ方式の3つの方式をリストアップし、これらを比較するための性能検証を行った。検証の結果、ファイル受け渡し方式を採用することとした。

③性能向上を目指したフレームワークの開発

ファイル受け渡しの方式を前提に、ファイル入出力を行うJavaバッチの性能を向上させるフレームワークを開発した。

④性能向上効果の検証

性能向上のためのフレームワークを適用した上で、再度Javaバッチの性能を測定した。フレームワークの効果により、COBOLバッチと同等以上にまでJavaバッチの性能を向上させることが可能なことがわかった。

以下、これら性能向上への取り組みについて詳しく説明する。

(2) 現状のJavaバッチ性能の把握

筆者らは、まず、現状のJavaバッチ性能の把握から始めることとした。Javaバッチの性能測定を行う上で、考慮したポイントは以下の通りである。

- COBOLバッチを比較対象とすること
  現在でも多くのバッチ処理はCOBOL言語で実装されている。よって、Javaバッチの性能をCOBOLバッチとの性能比で示すことが多くのバッチ開発者にとってわかりやすい指標となると判断した。
- 指標として適切なバッチアプリケーションを用いること
  実プロジェクトでサーバのサイジングに用いられているバッチアプリケーションを借用し、性能測定の対象とした。このアプリケーションは典型的なバッチ処理を実装したものであり、データ量も実システムを想定したものとなっている。アプリケーションはCOBOL言語で実装されていたため、これをJavaバッチへ移植した上で性能を比較した。
- Javaバッチへの移植においてプログラムロジックを変更しないこと
  ありのままのJava実行環境の性能を測定することが目的であるため、移植において性能を向上させるような特別な工夫は行わなかった。つまり、「ファイル受け渡しからメモリ受け渡しへ変更する」ようなプログラムロジックの変更は行わなかった。よって、これまでのCOBOLバッチと同様の設計手法でJavaバッチを開発した場合の性能指標が得られることになる。ただし、最新のJava実行環境が備える性能向上技術（NewIO等）については、これを利用した上で移植を行った。

以上のポイントを考慮した上で、COBOLバッチ、Javaバッチを用意し性能測定を行った。性能測定の環境は以下の通りである。

- ソフトウエア
  COBOL：COBOL2002
  Java：SUN JRE1.4.2
  OS：Windows Server 2003

- ハードウエア
  機種：HA8000/130
  CPU：Xeon 3.4GHz × 2
  メモリ：2GByte
  ディスク：36G × 3

性能測定の結果、現状のJavaバッチの性能はCOBOLバッチと比較して、以下のようになった。（処理時間は多重度が1の場合）

処理時間：2.56倍
スループット：0.4倍

予想通りJavaバッチの性能が低いことがわかったが、その差は3倍以内であり、性能向上のための工夫によりこの課題を解決でき

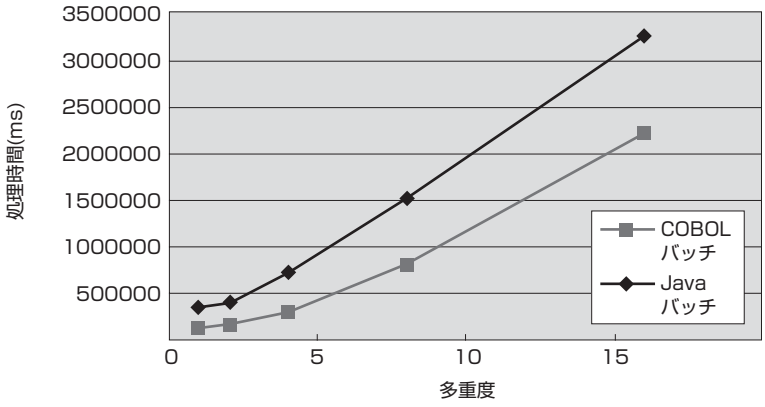


図1　現状の処理時間

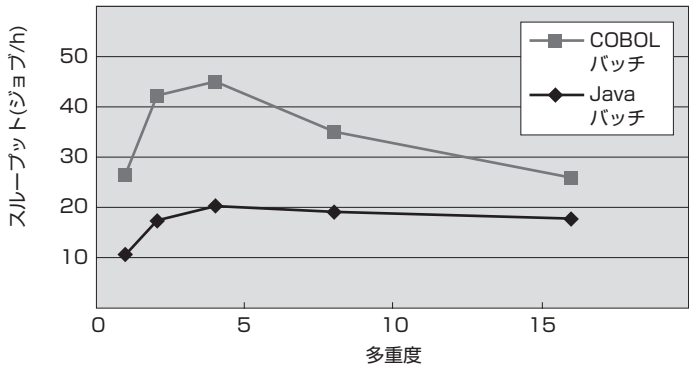


図2　現状のスループット

る可能性が高いと判断し、以降の取り組みを行うこととした。

（3）性能向上のための実装アーキテクチャの選定

①実装アーキテクチャの検討

次に、性能向上を目指したJavaバッチの実装アーキテクチャの検討を行った。アーキテクチャ選定のための候補として、一般によく知られているPOSAアーキテクチャパターン分類（『ソフトウエアアーキテクチャ』F.ブッシュマン他著）を用いることとした。この中からバッチ処理に適しているとされる「Pipes and Filtersパターン」をベースに検討を行った。Filter部分は、業務処理そのものであり一概に性能向上のための実装方式を規定することができない。一方、Pipe部分は、データを処理から処理へと受け渡す汎用的な部分であり、性能向上のための実装方式を規定することが可能である。そこで、Pipe部分の実装方式の候補をリストアップした上で、実機上での性能検証を行い、最適な実装方式を選定することにした。Pipe部分の実装方式の候補として、以下の3つをリストアップした。

- ファイル方式
  中間ファイルを使って、データを受け渡す方式。
- RDB方式
  RDB上のテーブルを使って、データを受け渡す方式。
- メモリ方式
  メモリ上のオブジェクトを使って、データを受け渡す方式。

②実装方式決定のための性能検証

Pipe部分の実装方式を選定するために、Pipe部分にファイル、RDB、メモリを利用した3つのパターンのプログラムを作成し、性能測定を行った。

また、バッチ処理において必ず必要となるソート処理をPipe部分の機能として捉えて、Pipe部分でソート処理を行うパターンのプログラムも作成し、合わせて性能測定を行った。それぞれソート処理は以下のように実装した。

- ファイル方式
  ソートユーティリティを外部コマンドとして呼び出し、ソート処理を実行。

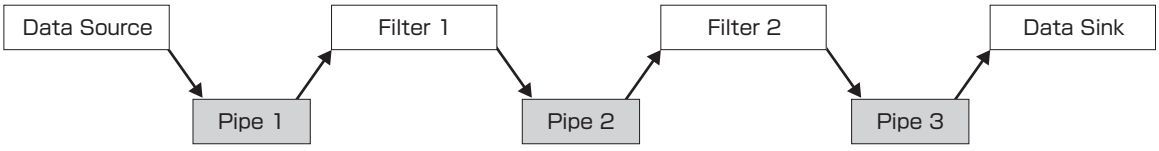


図3　Pipes and Filtersパターン

● RDB 方式
SQL の「ORDER BY 句」を利用し、データ読み込みと同時にソート処理を実行。
● メモリ方式
Java 実行環境が備える Collection ライブラリを利用し、ソート処理を実行。

これらの性能検証パターンを、図 4 に示した。

性能測定の結果を、表 1、表 2 に示す。

③ 実装アーキテクチャの選定

Pipe 部分の実装方式決定のための性能検証の結果を受け、以下のように考察した。

● RDB 方式の性能は非常に低い
ほとんどのパターンで、RDB 方式の性能は

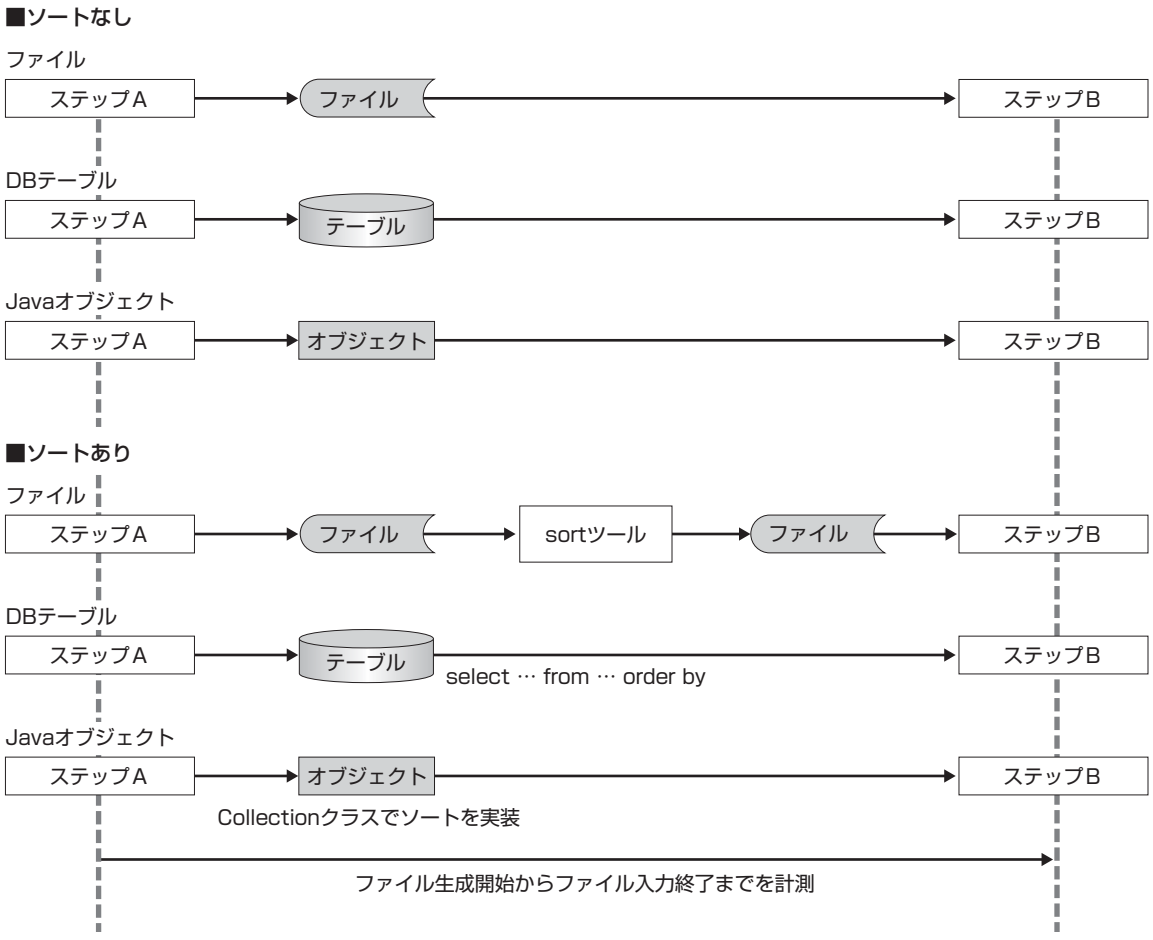


図 4　実装方式検証ジョブパターン

サイズ	種別／多重度	1	2	4	8	16
10K	ファイル	0.011	0.005	0.004	0.006	0.006
	DB	0.138	0.196	0.355	0.720	1.418
	Javaオブジェクト	0.004	0.003	0.003	0.003	0.003
100K	ファイル	0.050	0.074	0.056	0.064	0.074
	DB	0.447	0.625	1.068	2.159	4.682
	Javaオブジェクト	0.043	0.040	0.052	0.048	0.043
1M	ファイル	0.330	0.496	0.938	1.620	2.273
	DB	3.433	5.372	10.420	19.543	40.505
	Javaオブジェクト	0.257	0.397	0.378	0.466	0.436
10M	ファイル	3.100	4.716	8.856	16.483	32.635
	DB	34.868	56.790	123.073	230.793	545.967
	Javaオブジェクト	2.572	3.484	5.546	5.201	8.718
100M	ファイル	30.902	40.520	78.770	167.556	512.896
	DB	462.740	711.766	1249.145	2831.931	5663.862
	Javaオブジェクト	21.433	46.133	OutOfMemoryExceptionで計測不能		

表1　処理時間（ソートなし）

サイズ	種別／多重度	1	2	4	8	16
10K	ファイル	0.056	0.071	0.346	0.507	1.337
	DB	0.138	0.178	0.333	0.706	1.470
	Javaオブジェクト	0.005	0.004	0.003	0.003	0.003
100K	ファイル	0.113	0.148	0.367	0.879	2.200
	DB	0.452	0.635	1.057	2.132	4.726
	Javaオブジェクト	0.037	0.045	0.061	0.053	0.049
1M	ファイル	0.501	0.849	1.009	2.395	3.802
	DB	3.448	5.586	10.453	19.797	46.810
	Javaオブジェクト	0.267	0.406	0.431	0.446	0.593
10M	ファイル	4.194	6.430	11.358	17.350	24.045
	DB	42.918	82.265	161.568	334.432	937.767
	Javaオブジェクト	2.619	4.535	7.825	9.256	9.682
100M	ファイル	30.742	46.668	90.271	334.620	1130.288
	DB	593.934	1077.847	1976.572	3588.000	7175.000
	Javaオブジェクト	22.361	48.292	OutOfMemoryExceptionで計測不能		

表2　処理時間（ソートあり）

他のファイル方式、メモリ方式に比べて非常に低く（約 1/10）なっている。RDB が得意とするソート処理を含むパターンの場合でも、この傾向は変わらない。

- ファイル方式とメモリ方式の性能差は小さい
  OS 側のファイルキャッシュが有効に働くため、ファイル方式とメモリ方式との性能にはほとんど差がない。

- データ量増加に対応できるのはファイル方式
  メモリ方式は、物理メモリ量の制約を受けるため、データ量と多重度の増加に対応できない。

以上の考察から、Pipe 部分の実装方式はファイル方式が最適であると判断できる。まとめると、性能向上を目指した Java バッチの実装アーキテクチャには、「Pipe 実装にファイルを用いる Pipes and Filters アーキテクチャ」が最適であるとの結論に達した。

（4）性能向上を目指したフレームワークの開発

①RI（Record Item）フレームワークの開発

Pipe 部分の実装にファイルを用いる Pipes and Filters アーキテクチャにおいては、ファイル入出力に伴うデータ変換処理を削減することが性能向上のポイントとなる。また、大量データを扱うバッチ処理の特性から、オブジェクト生成処理を削減することも性能向上のポイントとなる。

しかし、これらの性能向上のためのポイントを踏まえて設計を行うことは難しく、また、開発者ごとのバラツキも出やすい。そこで、筆者らは、あらかじめ性能向上のための工夫を取り入れた RI（Record Item）フレームワークを開発した。RI フレームワークでは、レコード形式でのデータ入出力とデータ保持の機能を提供している。

②RI フレームワークの性能向上手法

RI フレームワークでは、以下の 2 つの性能向上手法を用いて Java バッチの性能向上を図っている。

- バイト列のままデータを保持する
  通常の Java プログラミングでは、データレコードを入力するたびに個々のデータ項目に対するオブジェクトを生成する。バッチ処理の場合は扱うデータ量が非常に大きいため、このような方法ではオブジェクト生成の回数が極端に増え、性能低下を招く。RI フレームワークでは、データレコードを格納するオブジェクトを用意しておき、データレコード入力時には内部のバイト配列のみを書き換える。これにより、データ量に関わらず、オブジェクト生成を一度だけ行えばよい仕組みとなっている。

●必要な部分のみ数値・文字列変換する

通常のJavaプログラミングでは、データレコードを入力するたびに個々のデータ項目に対する数値オブジェクト、文字列オブジェクトを生成する。つまり、入力のタイミングですべてのデータ項目に対して数値、文字列変換が行われる。ところが、バッチ処理の対象はデータレコードの一部のみであることが多く、すべてのデータ項目に対して変換処理を行うことは非常に無駄が多い。

RIフレームワークでは、データレコードをバイト配列のまま保持しておき、実行時に必要となったデータ項目のみを数値、文字列変換する方法を取っている。

また、同一データ型の項目同士の比較は変換処理を行わずにバイト列のまま比較する等の方法によっても、変換処理を削減している。

③RIフレームワークの機能

RIフレームワークの機能の一覧を表3に示す。

④RIフレームワークによる開発イメージ

RIフレームワークでは、データレコードをJavaクラスとして定義する。これを、レコード定義クラスと呼んでいる。レコード定義クラスのコーディング例を図6に示す。レコード定義クラスの利用方法を示すコーディング例を図7に示す。

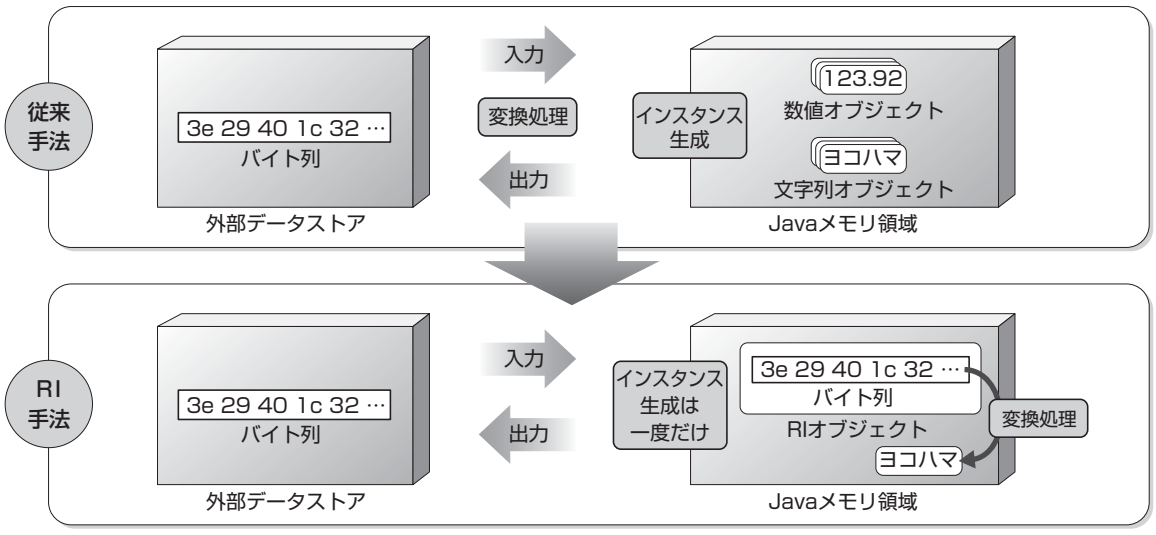


図5　RIフレームワークの性能向上手法

クラス	機能	説明
データ保持クラス	バイト配列保持	データをバイト配列で保持することで、文字コード、フォーマット変換を排除。
	データ構造	階層型、繰り返し型のデータ構造をサポートし、一般的な業務データ構造にフィット。
	JavaBeansインターフェイス	RecordItemクラスを用いながら、オンライン処理と同一のインターフェイスを提供する。
中間ファイルクラス	ファイルアクセス	Fileクラスのラッパーとしてファイルアクセス機能を提供する。
	RecordItem I/F	RecordItemクラス単位での入出力機能を提供する。
DBアクセスクラス	RecordItem I/F	RecordItemクラス単位でのDBアクセス機能を提供する。

表3　RIフレームワークの機能一覧

```
public class RI_Date extends RecordItem {
    public RI_PicX yy = new RI_PicX(2);
    public RI_PicX mm = new RI_PicX(2);
    public RI_PicX dd = new RI_PicX(2);
    public RI_Date() {
        this.add(this.yy);
        this.add(this.mm);
        this.add(this.dd);
    }
}
```

図6　レコード定義クラス例

```
public void testRecordItem() {
    // RIのインスタンス生成と初期化
    RI_Date date1 = new RI_Date();
    date1.initializeRecord();
    RI_Date date2 = new RI_Date();
    date2.initializeRecord();

    // RIへのデータのセット
    date1.setString("060718"); // レコードを一度に
    date2.yy.setString("06"); // 項目別に

    // RIとRIの比較（バイナリ比較）
    if (date1.yy.equals(date2.yy)) {
        // RIからRIへのコピー（バイナリコピー）
        date1.mm.moveTo(date2.mm);
        date1.dd.moveTo(date2.dd);
    }
}
```

図7　レコード定義クラス利用例

(5) 性能向上効果の検証

①性能測定の内容

RIフレームワークによる性能向上効果を検証するため、フレームワーク適用後のJavaバッチを用意し再度、性能測定を行った。今回も、前回と同様にCOBOLバッチを比較対象とするため、COBOLバッチ、Javaバッチ両方のバッチプログラムを作成し、これらの性能を測定した。

なお、後述する生産性向上のためのFCIフレームワークについても、同時に適用した上でJavaバッチの性能測定を行っている。これは、FCIフレームワークにも一部性能向上が見込まれる機能が含まれるからである。

測定対象としたバッチプログラムは、それぞれ以下のものである。

- COBOLバッチ

  「現状のJavaバッチ性能の把握」で用いたCOBOLバッチから一部を取り出し、測定対象とした。(今回は、JavaバッチをCOBOLバッチからの単純移植ではなく、再構築により開発することになる。よって、開発期間の短縮を目的に、対象とするCOBOLバッチの範囲を絞った。ただし、典型的なバッチ処理を含むように範囲設定した。)

- Javaバッチ

  COBOLバッチの外部仕様をリバースエンジニアリングにより抽出した上で、この外部仕様に基づくJavaバッチを、フレームワークを利用し再構築した。

COBOLバッチ、Javaバッチの処理フローを図8に示す。

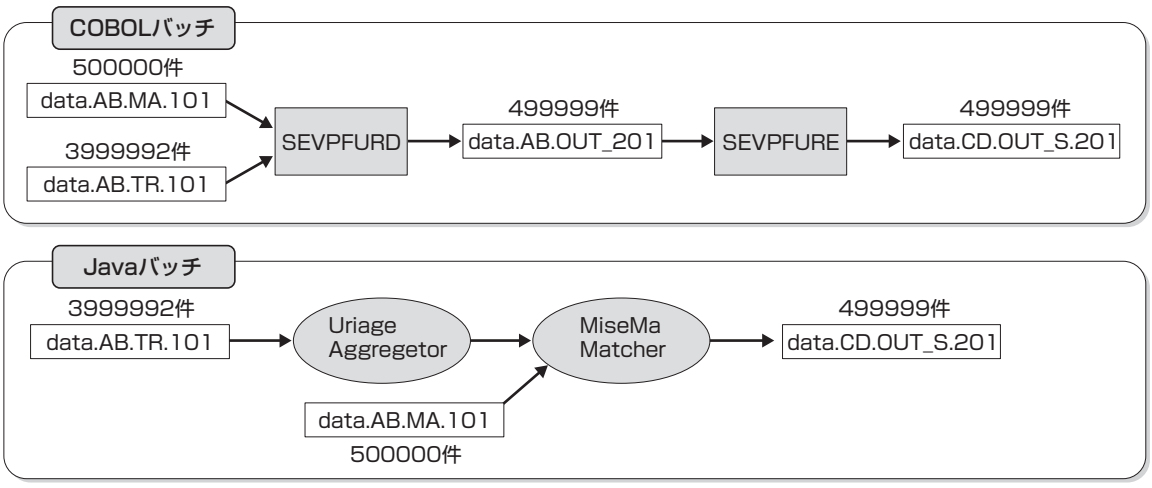


図8　処理フロー

Javaバッチは、中間ファイル（data.AB.OUT_201）を利用しない処理フローとなっている。これは、後述するFCIフレームワークの中間ファイル削除機能の効果である。

バッチプログラムの入出力ファイルを表4に示す。入出力ファイルの合計サイズは約1GBであり、実システムを想定したデータ量となっている。

性能測定に利用したハードウエア、ソフトウエアを図9に示す。今回も、同一のハードウエア上で、COBOLバッチ、Javaバッチの性能測定を行った。

②性能測定の結果

性能測定の結果、Javaバッチの性能はCOBOLバッチと比較して以下のようになった。（処理時間は多重度が1の場合）

処理時間： 1/3倍
スループット： 4倍

よって、性能向上を目指したフレームワークの適用により、Javaバッチの性能はCOBOLバッチと同等以上にまで向上することがわかった。

また、フレームワークに組み込まれた性能向上手法の効果を個別に測定した結果を表5に示す。これによると、オブジェクトの再利用（オブジェクト生成処理の削減）が、Javaバッチの性能向上に最も効果があることがわ

I/O	入出力ファイル	データ内容	件数	レコード長	ファイルサイズ(KB)
I	data.AB.MA.101	店マスタ	500000件	1200	585,938
I	data.AB.TR.101	売上データ	3999992件	20	195,313
O	data.AB.OUT_201	店別売上中間データ（COBOLのみ）	499999件	256	125,000
O	data.CD.OUT_S.201	店別売上データ	499999件	256	125,000

表4　入出力ファイル

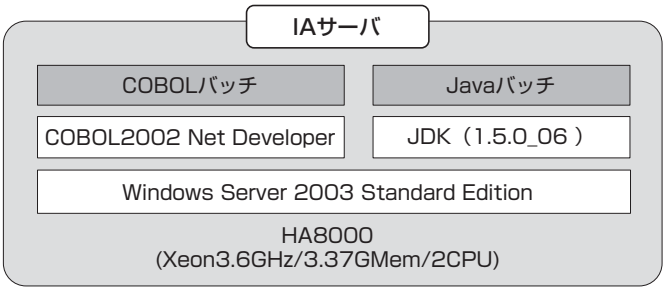


図9　性能測定環境

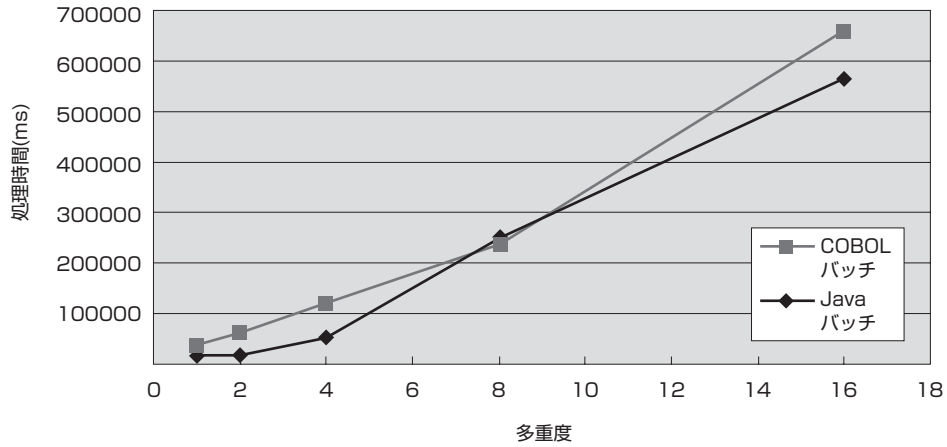


図10　性能向上後の処理時間

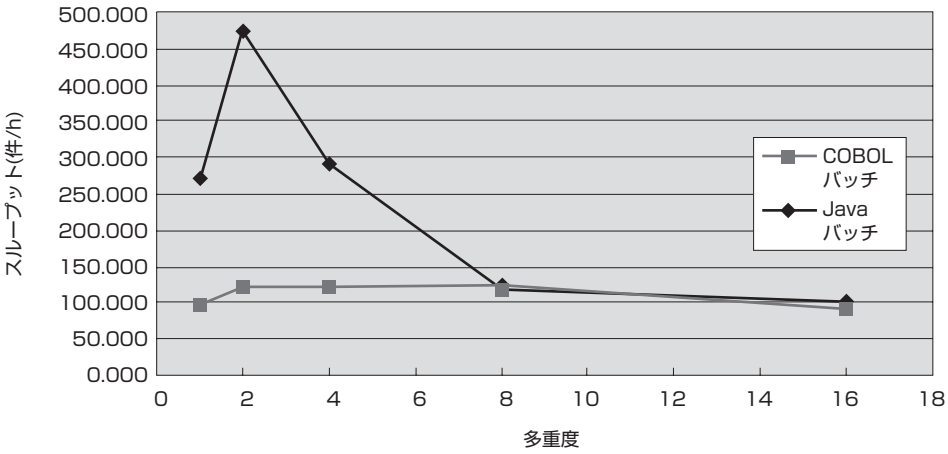


図11　性能向上後のスループット

カテゴリ	性能向上手法	削減時間	削減率
RIフレームワーク	オブジェクトの再利用	−67.2秒	−83.5%
	データ変換処理の削減	−23.8秒	−64.2%
FCIフレームワーク	中間ファイルI/Oの削除	−1.7秒	−13.0%
合計		−92.7秒	−88.7%

表5　手法ごとの性能向上効果

かった。

③性能向上効果のまとめ

性能向上のためのフレームワークを適用する前のJavaバッチの性能は、COBOLバッチと比較して、(処理時間は多重度が1の場合)

処理時間： 2.56倍
スループット： 0.4倍

であったが、フレームワークの適用により、

処理時間： 1/3倍
スループット： 4倍

にまで向上した。つまり、フレームワークの適用により約10倍の性能向上効果が得られたことになる。

また、COBOLバッチと同等以上の性能を発揮できることが確認されたことで、Javaバッチの性能面での不安はほぼ解消されたと考える。

4．Javaバッチの生産性向上への取り組み

(1) 生産性向上への取り組みの概要

Javaバッチの生産性向上を目指して以下のような取り組みを行った。

①開発アーキテクチャの選定

Javaバッチの生産性向上を目指した設計開発手法を検討した結果、設計モデルにDFD(データフローダイアグラム)を採用しDFDの段階的詳細化により設計を行う手法を選定した。

②生産性向上を目指したフレームワークの開発

DFDを中心とした開発アーキテクチャを前提に、生産性向上を目指したフレームワークを開発した。

③導入支援ドキュメントの整備

フレームワーク導入を支援し、生産性向上に繋げるための各種ドキュメントを検討し、その一部を作成した。

④最新の開発技術導入の可能性

Java言語でバッチ開発することで可能となる最新の開発技術／ツールの導入について検討した。

以下、これらの生産性向上への取り組みについて詳しく説明する。

(2) 開発アーキテクチャの選定

Javaによるオンラインアプリケーションの開発では、生産性、品質の向上を目指したフレームワークを使って開発を行う方法が一般化している。筆者らは、バッチアプリケーションにおいても同様のアプローチが有効であると考え、Javaバッチ開発の生産性向上を目

指したフレームワークを開発することにした。

そこで、まずJavaバッチの開発アーキテクチャ（設計開発の手法）について検討を行った。検討にあたって、以下の2つの要件を重視した。

- 設計手法がシンプルでわかりやすいこと
  バッチ処理は、データの流れとデータに対する処理の組み合わせとして捉えるとわかりやすい。バッチ処理の設計モデルでは、これを自然に表現できる必要がある

- 設計情報のトレーサビリティが確保されること
  上流工程と下流工程の設計情報の紐付けを容易に把握できることが重要である。これにより、上流工程で発生した仕様変更を下流工程の設計情報に迅速に反映させることが可能となる。また、下流工程（例えば、連結テスト中）で発見された仕様の不整合を上流工程の設計情報に遡って修正することが容易に行えるようになる。

上記の要件を重視し検討を行った結果、Javaバッチの開発アーキテクチャを以下のように決定した。

- 設計モデル：DFDを採用
  バッチアプリケーションの設計モデルとしてDFDを一貫して採用する。DFDではデータの流れに着目して設計情報をモデル化するため、バッチ処理を自然に表現することができる。

- 設計手法：DFDの段階的詳細化により設計を行う
  上流工程から下流工程まで、DFDを段階的に詳細化することにより設計を行う。これにより、DFDをキーとして設計情報のトレーサビリティを確保することが可能となる。段階的詳細化のイメージを、図12に示す。

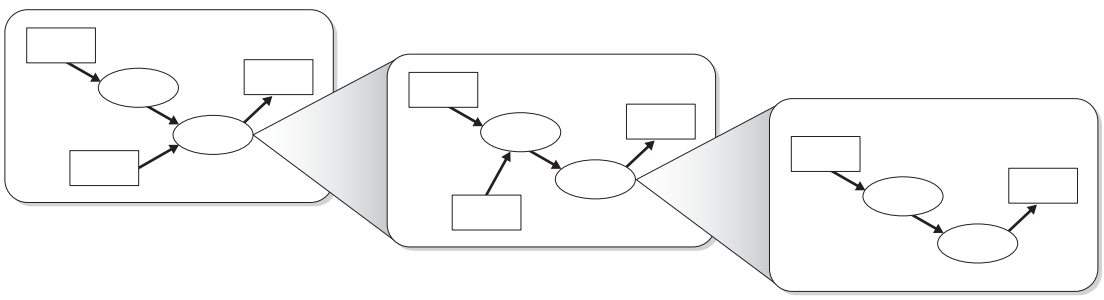


図12　DFDの段階的詳細化

●開発手法：DFD構成要素と開発モジュールとの対応付け

下流工程のDFDモデル図の構成要素が、そのまま開発モジュールに対応付けられるようにする。これにより、モジュール分割のための設計を別途行う必要がなくなる。また、設計情報だけでなく開発モジュールまでも含めたトレーサビリティの確保が容易に行えるようになる。

(3) 生産性向上を目指したフレームワークの開発

①FCI (Flow Control Injection) フレームワークの開発

DFDを中心とした開発アーキテクチャを前提に、Javaバッチ開発の生産性向上を目指したFCI (Flow Control Injection) フレームワークを開発した。なお、FCIフレームワークの実行制御手法については特許出願中である。

②FCIフレームワークの特徴

FCIフレームワークの特徴を以下に示す。

●DFDからクラス設計へのダイレクトマッピング

FCIフレームワークを前提とした設計では、入力処理、出力処理、マッチング処理、集約処理 (ブレーク処理) の4つの処理のみから構成される状態となるまで、DFDを段階的に詳細化する。詳細化が完了したDFDのモデル要素は、そのまま実装すべきクラスに対応する。つまり、FCIフレームワークを利用した開発では、DFDの詳細化を行うことでクラス設計までの設計作業が完了することになる。

●業務ロジックとフロー制御ロジックの分離

フロー制御ロジック (データの受け渡し制御やマッチング制御等) は、FCIフレームワークが提供する。よって、バッチアプリケーションの開発者は純粋な業務ロジックのみを実装するだけでよい。これに加えて、FCIフレームワークではフロー制御ロジックを実行時に交換可能な仕組みとしている。これを、フロー制御ロジックの注入 (Flow Control Injection) と呼んでいる。フロー制御ロジックを変更するにあたっては、バッチアプリケーション側の修正は一切不要である。

現在、フロー制御ロジックとして、シングルスレッド制御とマルチスレッド制御の2つを提供しているが、将来はグリッド実行制御 (複数のサーバによる分散処理) を提供することも可能である。

●不要な中間ファイルの削除

フロー制御ロジックをFCIフレームワークが提供することにより、プログラムのメインループはアプリケーションからフレームワーク側に移動することになる。FCIフレ

ームワークのフロー制御ロジックでは、処理から処理へ直接データの受け渡しを行うため、データの受け渡しだけを目的とした不要な中間ファイルを削除することができる。このイメージを図13に示す。
また、これによって、DFD上ではプロセスとプロセスの間を直接データの流れとして結ぶことができ、シンプルなDFDを使って設計することが可能となる。
ただし、ジョブのリランポイントとするために中間ファイルを利用している場合や、中間ファイルでソート処理を行う場合は、これらを削除することはできない。この場合は、この中間ファイルの前後でバッチアプリケーション（DFD）を分離することになる。

③FCIフレームワークを利用した開発の概要

FCIフレームワークを利用したバッチアプリケーション開発の概要を以下に示す。

a バッチ処理のDFDによるモデル化

バッチアプリケーションが対象とする業務処理をDFDによりモデル化する。

b プロセスの詳細化

DFD中のプロセスに着目し、段階的に詳細化したDFDを作成する。

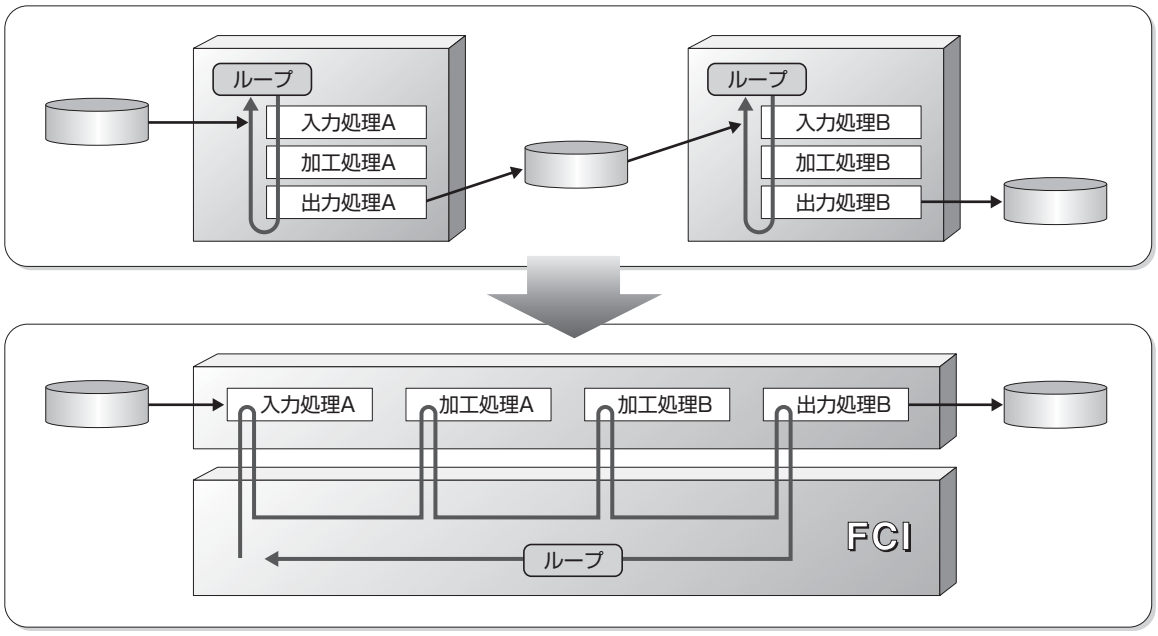


図13　不要な中間ファイルの削除

c 集約処理（ブレーク処理）の詳細化

複数レベルでのブレーク処理を含む複雑な集約処理をさらに詳細化する。

d 詳細化DFDを元に設計、コーディング

詳細化が完了したDFDをもとに、詳細設計、コーディングを行う。

e フロー制御ロジックの注入

完成したバッチアプリケーションに対してフロー制御ロジックを注入する。

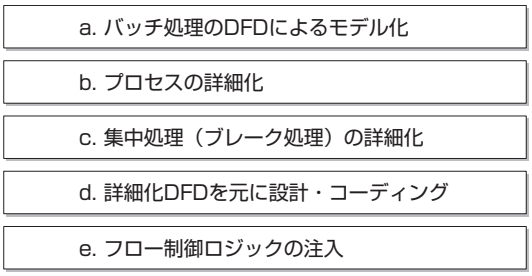


図14　FCIを利用した開発の流れ

④FCIフレームワークを利用した開発の流れ

a バッチ処理のDFDによるモデル化

バッチアプリケーションを含む業務処理全体をDFDによりモデル化する。このDFDでは、概要設計もしくは外部設計レベルの設計情報を整理し表現する。DFDの作成にあたっては、表6で示すような一般的な記法を用いる。

概要レベルDFDの例を図15に示す。

b プロセスの詳細化

DFD中の特定のプロセスに着目し、プロセスの内部で実行される業務処理を詳細化した別のDFDを作成する。これをDFD中の各々のプロセスについて行うことで、DFDの段階的詳細化を行う。最終的に、DFDの構成要素が、入力処理、出力処理、マッチング処理、集約処理（ブレーク処理）の4つの処理のみの状態となるまで詳細化を行う。

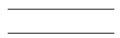
名称	シンボル	説明
源泉と吸収		DFDにデータが流れ込む元と、DFDからデータが流れ出す先を示す。 利用者や外部システムが相当する。
データストア		データの一時的な保存場所を示す。 ファイルやデータベースが相当する。
プロセス		データに対して適用する業務処理を示す。
データフロー		データの流れを示す。

表6　DFDの記法

4つの処理の意味を以下に示す。

●入力処理
ファイルやデータベースからの単純な入力処理を示す。源泉のシンボルを利用して表現する。データストアからの入力は、データストアのシンボルを使わず、この入力処理として表現する。つまり、データストアを含むDFDはデータストアを境として別のDFDに分離されることになる。

●出力処理
ファイルやデータベースへの単純な出力処理を示す。吸収のシンボルを利用して表現する。データストアへの出力は、データストアのシンボルを使わず、この出力処理として表現する。

●マッチング処理
2つのストリーム（データの流れ）から入力されたデータを特定のキー（マッチングキー）により突き合わせることによって行う業務処理を示す。プロセスのシンボルを利用して表現する。（例：売上データの中の店コードをキーに店マスタを突き合わせて店名を追加する処理）
突き合わせ元のストリームをトランザクションストリーム、突き合わせ先のストリームをマスターストリームと呼び、2つのストリームを区別する。DFD上では、これらを区別するためにデータフローの矢印の近くに「TR Stream」、「MA Stream」と記述する。

●集約処理（ブレーク処理）
1つのストリーム（データの流れ）から入力されたデータを特定のキー（ブレークキー）で集約することによって行う業務処理を示す。プロセスのシンボルを利用して表現する。（例：売上データの中の地区コードをキーに地区ごとの売上合計を集計する処理）

詳細化が完了したDFDの例を図16に示す。このDFDは、図15の中にある「売上集計プロセス」を詳細化したものである。

詳細化が完了したDFDには、FCIフレームワークのモデル要素との対応が分かり易い

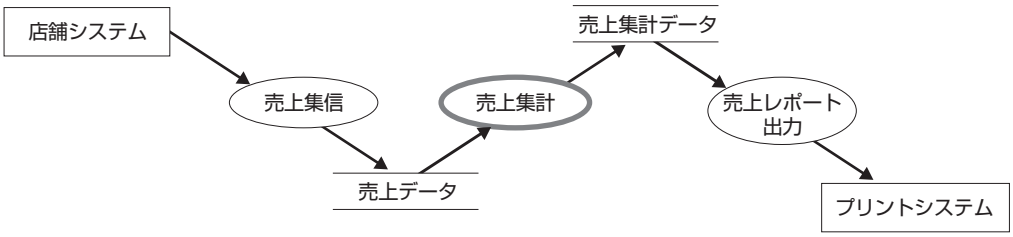


図15　概要DFDの例

ように以下のステレオタイプ（モデル要素の種類）を追記する。

- ProcessingModel
  詳細化が完了したDFD全体を示すモデル要素。FCIフレームワークでは、このProcessingModelがバッチアプリケーションの作成単位となる。
- Source
  入力処理を表すモデル要素。
- Sink
  出力処理を表すモデル要素。
- Matcher
  マッチング処理を表すモデル要素。
- Aggregator
  集約処理（ブレーク処理）を表すモデル要素。

ステレオタイプの追記が完了したDFDの例を図17に示す。

c 集約処理（ブレーク処理）の詳細化

プロセスの詳細化が完了したDFDの中に集約処理が含まれる場合、さらに集約処理の詳細化を行う。

FCIフレームワークでは、複数レベルのブレーク処理（複数のブレークキーを使った多段階の集約処理）から成る集約処理の設計を単純化するため、集約処理をレベルごとのブレーク処理に分割して設計する方法をとっている。この方法に従って、集約処理の詳細化を行う。

例を、図18に示す。この例では、売上集約処理（Aggregator）を、全体、地区、店の3つのレベルごとのブレーク処理に分割している。

レベルごとの集約処理を表すモデル要素をAggregateProcessorと呼ぶ。

d 詳細化DFDを元に設計、コーディング

プロセスの詳細化、集約処理の詳細化が完

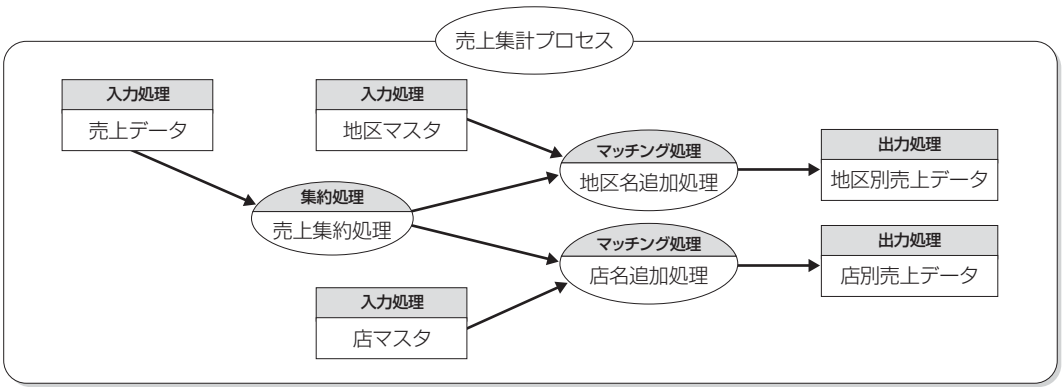


図16　詳細DFDの例

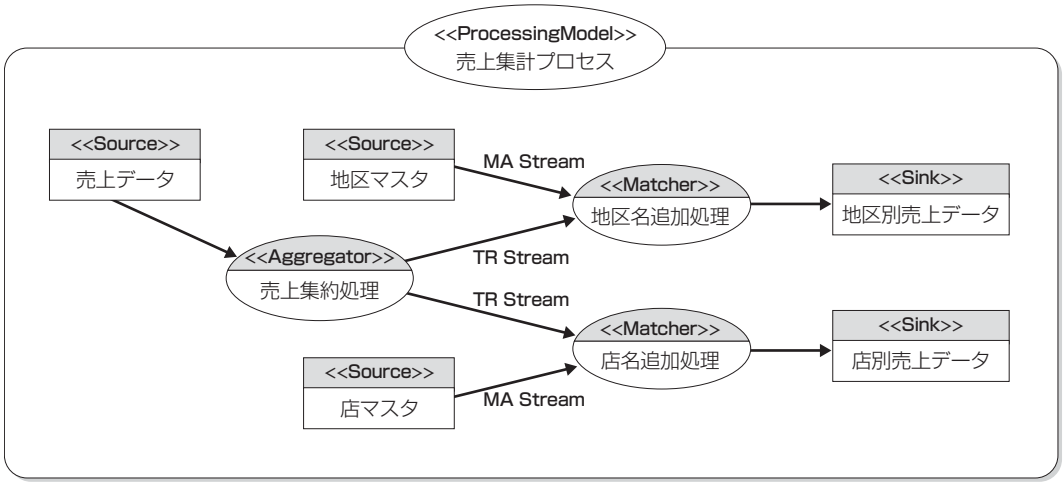


図17　ステレオタイプ追記 DFD の例

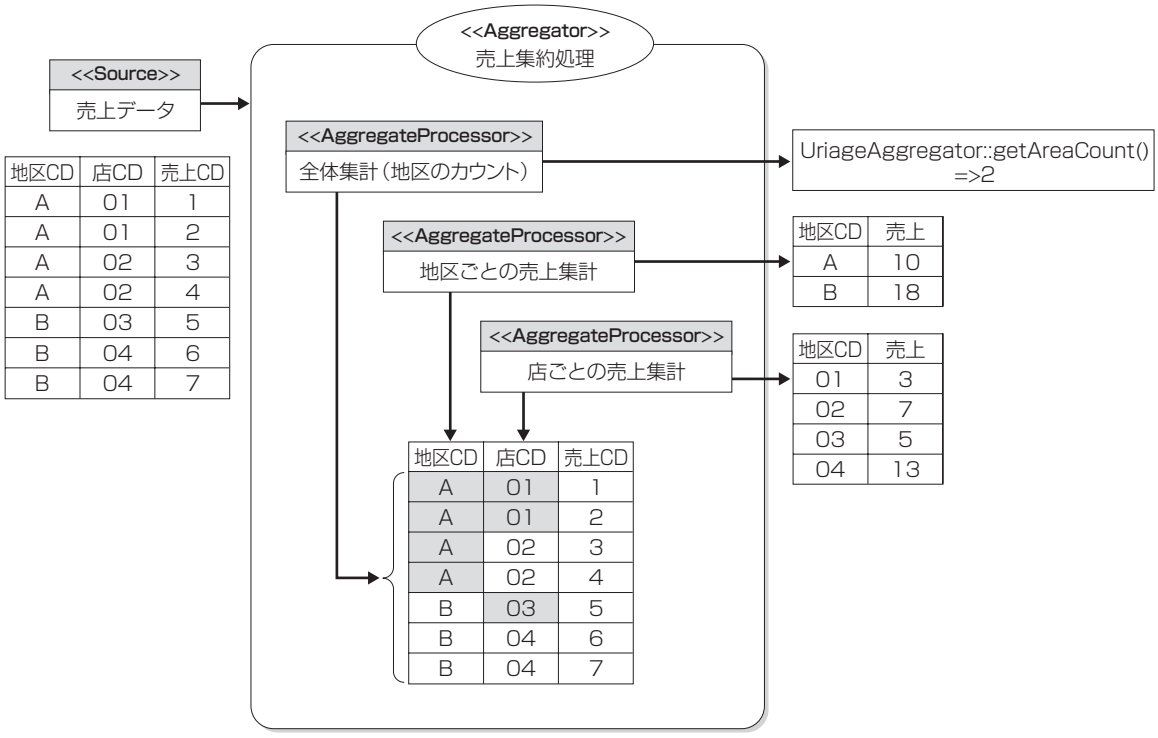


図18　集約処理詳細化の例

了したDFDをインプットに、バッチアプリケーションの詳細設計、コーディングを行う。

詳細化が完了したDFDのモデル要素は、FCIフレームワークを利用する上で作成すべきクラスと1対1に対応している。つまり、DFDの詳細化が完了した時点で、クラス分割（モジュール分割）までの設計が完了していることになる。

引き続き、これらのクラスごとに詳細設計、コーディングを行う。コーディング時には、これらのクラスごとにFCIフレームワークが提供するスーパークラスを継承し、業務ロジックを実装する。

e フロー制御ロジックの注入

最後に、完成したバッチアプリケーションに対してフロー制御ロジックを注入する。

具体的には、バッチアプリケーションを表すProcessingModelクラスのインスタンスを生成し、これを引数として適切なフロー制御クラスのdoTask（）メソッドを呼び出す。これにより、選択したフロー制御ロジックに従ってバッチアプリケーションの実行が行われる。

現在、FCIフレームワークでは、以下の2つのフロー制御クラスを提供している。

- SingleThreadFlowController
  入力処理で読み込まれたデータごとに、シングルスレッドで後続の集約処理、マッチング処理、出力処理を呼び出す方式のフロー制御を行う。

- MultiThreadFlowController
  入力処理、集約処理、マッチング処理、出力処理のすべての処理をマルチスレッドで実行しながらデータの受け渡しを行う方式のフロー制御を行う。

フロー制御ロジックの抽入方法のコーディング例を図19に示す。

（4）導入支援ドキュメントの整備

Java言語によるバッチ開発を現場へ導入して生産性を向上させるためには、単にフレームワークを提供するだけでなくフレームワークの導入を支援する各種ドキュメントを整備し、提供する必要がある。

今回、筆者らは、性能検証プロジェクトの

```
シングルスレッドの場合
        singleThreadFlowController.doTask( processingModel )

マルチスレッドの場合
        multiThreadFlowController.doTask( processingModel )
```

図19　フロー制御注入のコーディング例

中でRIフレームワーク、FCIフレームワークを利用したJavaバッチの開発を行った。この中で、フレームワークの適用を円滑に行うための基本設計書の雛型（テンプレート）を作成した。

今後、以下のような標準化ドキュメント・手順書を実プロジェクト適用の中で作成していく予定である。

●DFD作成標準化ガイド

概要レベルDFDの記述方法を標準化するためのガイドラインドキュメント。FCIフレームワークを利用した設計では、詳細レベルDFDの作成基準は明確であり、設計者間のバラツキは少ない。一方で、概要レベルDFDについてはゆるやかな基準しか規定していないため、設計者によって異なったレベルのDFDを作成してしまう可能性が高い。よって、概要レベルDFDの記述方法、記述レベルをプロジェクト内で標準化するためのDFD作成標準化ガイドが必要となる。

●DFDマッピング標準化ガイド

業務パターンからDFDへのマッピングを標準化するためのガイドラインドキュメント。DFDは自由度の高い記述方法であるため、同一の業務処理を対象としても、分析者、設計者によってまったく異なったモデルを作成してしまう場合がある。よって、対象業務群から業務パターンを抽出し、業務パターンごとのDFDの雛型を示すDFDマッピング標準化ガイドを利用した標準化が有効であると考えられる。

この標準化により、保守性が向上するとともにアプリケーション間のモジュールの共有化が促進され生産性の向上も期待できる。

●Javaバッチコーディング基準書

Javaバッチのコーディングルールを規定するための基準書。今回、筆者らは、Javaバッチの性能向上を目指したRIフレームワークを作成し、その効果を確認した。しかし、フレームワーク側でいくら性能を向上させても、アプリケーション側で性能を低下させるようなコーディングがあればその意味がなくなる。よって、Javaバッチコーディング基準書を用意し、性能を低下させるようなコーディングを防ぐための取り組みが必要である。

また、コーディング基準書によるソースコードの可読性向上や保守性向上の効果も重要である。

●Javaバッチ単体テスト手順書

Javaバッチの単体テスト方法を規定するための手順書。FCIフレームワークを利用したバッチアプリケーションの単体テスト工程においては、単体テストを行うモジュールの単位やそのテスト方法を規定するた

めの手順書が必要である。

（5）最新の開発技術導入の可能性

最新の開発技術／ツールの多くは、Java言語によるアプリケーション開発をターゲットとしたオープンソースプロジェクトから生み出されている。特に、統合開発環境であるEclipseと、Eclipseを基盤として動作する様々な開発支援ツールのプロジェクトでは、新たな開発技術への取り組みと導入が活発に行われている。

Javaバッチの開発にこれらの開発技術／ツールを導入することによって、さらなる生産性、品質の向上が期待できる。

今後、以下のような開発技術／ツールのフレームワークへの取り込みを予定している。

- テスト自動化ツール
  テスト自動化ツールとフレームワークを連携させ、単体テスト、連結テストの自動実行や回帰テストを実現する。

- AOP（アスペクト指向プログラミング）ツール
  デバッグ時のみ利用するデータダンプ機能などを、本体のプログラムを変更せずに後から追加できるようにする。

5．様々なJavaバッチ処理方式への対応

（1）J2SEバッチとJ2EEバッチ

Javaアプリケーションの実行環境にはJ2SE環境とJ2EE環境があり、各々の実行環境で動作するJavaバッチをJ2SEバッチ、J2EEバッチと呼ぶことにする。これらJavaバッチの特徴を以下に示す。

- J2SEバッチ
  J2SEバッチは、OS上のコマンドラインアプリケーションとして動作する。つまり、従来のCOBOLバッチと同様に運用管理ツールから起動し、管理することが可能である。よって、J2SEバッチは従来のバッチシステムの一部にJavaバッチを導入したり、段階的にJavaバッチに移行する場合に適した手法である。

- J2EEバッチ
  J2EEバッチは、J2EEアプリケーションサーバ上のアプリケーションとして動作する。よって、オンラインアプリケーションから非同期に実行されるディレードオンライン処理や、まとまったデータに対してアプリケーション処理を一括実行するセンターカット処理に適した手法である。
  J2EEバッチは、オンラインアプリケーションと共通の基盤を利用するため業務ロジックや共通部品等の再利用を行いやすいという特徴がある。

また、J2EE基盤が提供するクラスタリング機能によりスケールアウト構成をとることで、性能の向上も期待できる。

(2) J2SEバッチへの対応

すでに述べたように、J2SEバッチは従来のバッチシステムの一部にJavaバッチを導入したり、段階的にJavaバッチに移行する場合に利用される手法である。

よって、J2SEバッチは他のバッチプログラムと混在した環境で動作する必要がある。そこで、筆者らは、バッチプログラムとして広く利用されているCOBOLバッチと、Javaバッチの連携をサポートする機能を開発した。

具体的には、RIフレームワークにCOBOL特有のデータ型（符号付ゾーン形式やパック形式等）の入出力機能を追加した。これにより、COBOLバッチとJavaバッチとの間のデータ連携が容易に行えるようになる。

(3) J2EEバッチへの対応

J2EE基盤はオンラインアプリケーションのために設計された基盤であるため、バッチアプリケーションのための実行制御機能を提供していない。よって、そのままではJ2EEバッチを実現することができない。

そこで、筆者らは、J2EE基盤にバッチ実行制御機能を付加するミドルウエアを開発した。（現在、オブジェクトワークス/BT JOB Managerとして製品化）

本ミドルウエアの構成を図20に、各々の機能を以下に示す。

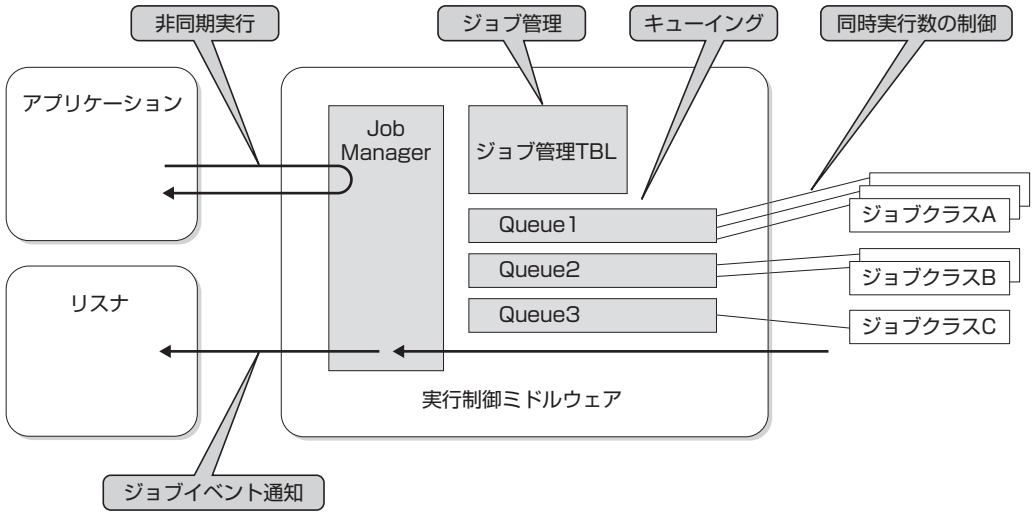


図20　J2EEバッチ実行制御機能

● ジョブの非同期実行機能
オンラインアプリケーションから、ジョブの非同期実行要求を受け付ける。オンラインアプリケーションに制御を戻し、バックグラウンドでジョブの実行を行う。

● ジョブ管理機能
ジョブの状態管理を行う。ジョブの状態には、実行待ち、実行中、正常終了、異常終了等がある。また、オンラインアプリケーションに対してジョブの状態を取得する機能を提供する。

● ジョブのキューイング機能
実行要求時に指定されたジョブキューに対してジョブのキューイングを行う。ジョブは、キューイングされた順に実行される。
また、ジョブキューの溢れ検知機能や、ジョブキューの閉塞・閉塞解除機能を提供する。

● ジョブの同時実行数制御
ジョブの同時実行数をジョブキュー単位に設定された最大値により制限する。これにより、ジョブキュー単位でCPUリソースを配分することができる。

● ジョブイベント通知機能
ジョブイベント（ジョブの状態変化）をオンラインアプリケーションに通知する。ジョブイベントには、ジョブの実行開始、ジョブの正常終了、ジョブの異常終了等がある。これにより、アプリケーション側でジョブの状態変化をトリガーにした処理を実行することが可能となる。

6．おわりに

現状では、オンラインアプリケーションをJava言語で開発していても、バッチアプリケーションは従来通りCOBOL言語で開発するという事例が多い。この主な理由は、Javaバッチ（Java言語によるバッチ処理）に対する性能面での不安が大きいためであると考えられる。一方で、生産性向上へのさらなる要求、COBOL技術者確保への不安、Java実行環境の性能向上を受けてJavaバッチへの期待は高まりつつある。このような状況を受けて、筆者らは、Javaバッチの実用化に向けた取り組みを行った。

まず、Javaバッチの性能向上を目指して、オブジェクト生成とデータ変換処理を削減するRI（Record Item）フレームワークを開発した。性能検証の結果、Javaバッチの性能をCOBOLバッチと同等以上にまで引き上げられることがわかった。

次に、Javaバッチの生産性向上を目指して、設計モデルにDFDを採用しDFDの段階的詳細化により設計を行う手法を採用したFCI（Flow Control Injection）フレームワークを開発した。

さらに、Javaバッチの様々な処理方式に対

応するための取り組みを行った。この中で、J2EE基盤にバッチ実行制御機能を付加するミドルウエアを開発した。

これらの取り組みにより、Javaバッチの実用化に対する目処を立てることができた。

一方、Javaバッチの本格的普及に至るまでにはいくつかの課題が残されている。例えば、今回は典型的なアプリケーションを対象に性能向上の取り組みを行ったが、今後は多様なアプリケーションを対象に性能向上のためのノウハウを蓄積していく必要がある。また、今回開発したフレームワークを活用し、生産性向上へと結びつけるためには、各種標準化ドキュメント、手順書を整備する必要がある。今後、実プロジェクトへの適用を重ねる中で、これらの課題に対する取り組みを行っていく予定である。

●参考文献●

［1］ 近代科学社『ソフトウエアアーキテクチャ』F.ブッシュマン他著　金澤典子他訳　2000年刊