White Paper
Micro Focus Visual COBOL
Micro Focus Enterprise Developer

デジタルトランスフォーメーションを見据えた COBOL バッチアプリケーションの最適化

~リホストや jBatch 等の JVM アプリ化によるモダナイゼーション~

基幹系システム、中でも大規模なバッチアプリケーションは、今でもメインフレーム等のコスト効率が低くなりがちなレガシー環境で稼働するものが多く残ります。本書ではバッチアプリケーションに着目し、ビジネスのデジタル化への投資力を強化させるべく、コスト効率の低い環境から脱却するための手段について論じます。既存の実証済みバッチアプリケーション資産を活かして新たな環境へ移行する 3 つの選択肢を紹介し、それぞれ詳しく解説します。

MICRO FOCUS

目次

1 はじめに

近年、様々な企業や団体がテクノロジーを駆使して過去にはなかった価値を商品やサービスにもたらす「デジタルトランスフォーメーション」に取り組んでいます。これはもはやスタートアップ企業やベンチャー企業に限った話題ではありません。大企業についても AI, Big Data, IoT, VR 等様々な先端技術を駆使して新たな接点やビジネスを創出することが企業競争力に大きく影響するようになりました。こうした中、メンテナンスコストが高止まりする基幹系システムの存在はデジタルビジネス活用に向けた活動の足かせになりつつあります。基幹系システム、中でも大規模なバッチアプリケーションは、今でもメインフレーム等のコスト効率が低くなりがちなレガシー環境で稼働するものが多く残ります。本書ではバッチアプリケーションに着目し、ビジネスのデジタル化への投資力を強化させるべく、コスト効率の低い環境から脱却するための手段について論じます。

第 3 章、第 4 章ではまず、レガシー環境上の資産になるべく手を入れずにコスト効率の高いクラウドを含めた最新プラットフォームへ移行させる手段について紹介します。

ビジネスのデジタル化について考えた場合、情報系の業務のみならず基幹系の業務も含めて取り組まなければ本当のイノベーションには乗り出せないのかもしれません。ビジネスのデジタル化を支える先進的な技術との親和性が高い技術の1つとして長年 JVM は注目を集めています。基幹系システムのオープン化に取り組まれる企業の多くは、オープン化の際 に Java 化による JVM 移行を検討されるかもしれません。しかし、長年メンテナンスしてきたアプリケーション資産の再構築は、再構築に係る費用のみならず再構築による品質低下のリスクも潜んでいます。実際、コスト超過、スケジュール超過、品質が満たせない等の理由でリリースに至れなかったケースも多く存在するようです。Micro Focus Visual COBOL が提供する JVM COBOL は COBOL 言語を JVM 言語として扱う機能です。本機能を利用すれば JVM 上でレガシー環境にて基幹系バッチアプリケーションとして稼動していた COBOL 資産を継続利用することも可能となります。JVM 向けのバッチフレームワークの代表格としては、Java EE 7 より導入された通称 jBatch、JSR-352（Batch Applications for Java Platform）が挙げられます。第5章ではこの jBatch という技術について簡単に整理します。その上で、JVM COBOL を活用し jBatch アプリケーションとして COBOL 資産を利用する方法を紹介します。

第7章では、ある簡単なサンプルアプリケーションを用いて本書で紹介する各方式とメインフレーム上での処理時間の計測比較を行います。パフォーマンスに関しては様々な要素に影響されるため、ここで紹介する計測結果による優劣が全てのケースに当てはまるわけではありません。参考指標の1つとしてお取り扱いください。

2 バッチアプリケーション移行の選択肢

そもそも本書で取り上げる「バッチアプリケーション」とはどのようなアプリケーションを指すでしょうか。一般的には、処理中にユーザーとの対話は発生せず、複数の手順から構成される処理を自動的に一括処理する処理方式を指します。多くの場合、これらの処理を通じて一定期間もしくは一定量溜めたデータを加工し、それに基づいてデータベースを更新します。扱う対象が大量データとなることからオンラインアプリケーションのように即時応答は要求されません。バッチアプリケーションとして運用される典型的な業務としては以下のようなものが挙げられます：

年、月、日等の単位で定期的に実行される定例業務

例）各月の請求書の作成、利子の反映、マスターファイルの更新
- 売上げ集計、決算処理等のように即時性が要求されない業務
- 金融シミュレーション等、膨大な CPU 時間を要する計算集約型業務

これらはいずれも企業活動を遂行する上で重要な業務です。特に年、月、日単位による締めの業務は企業問わず欠かせないものではないでしょうか。これらの業務が必要とされている限りバッチアプリケーションは運用し続ける必要があります。

しかし、業務上で必要であっても少なくとも以下の観点から長年運用を続ける筐体上で継続利用し続けることは得策ではありません。

- パフォーマンス
- コスト
- 開発生産性

バッチアプリケーションは大量の演算処理並びに IO 処理を行うので、実行時のサーバー資源への負荷は考慮する必要があります。そのため、サーバー資源に余裕のある夜間や休日に運用されることが多いです。しかし、近年ビジネスのグローバル化やアプリケーションのモバイル化等によりオンラインアプリケーションの稼動時間が拡大される傾向にあり、それに伴いサーバー資源に余裕のある時間帯そのものが短縮されています。更に IoT や SNS の発展に伴い企業が扱うデータ量そのものが増加の一途を辿っており、連動してバッチアプリケーションで扱うデータ量も増える可能性がないとは言い切れません。より短い時間で、より多くのデータを処理するハイパフォーマンスなバッチアプリケーションが求められます。

バッチアプリケーションに関しては今でもメインフレームやオフコン、もしくはいわゆるオープンレガシーと呼ばれる保守が打ち切られた UNIX サーバーや Windows サーバー上で動作することも少なくありません。SoE(System of Engagement) の領域のアプリケーションがコスト効率の高いクラウド環境等で運用される傾向にある中、SoR(System of Record) の領域であるからといってバッチアプリケーションの TOC のみを度外視することは難しくなってきています。

開発生産性についてもバッチアプリケーションについて SoE と比べると比較的劣る傾向にあります。例えば、SoE については IDE に搭載されたインテリセンスを駆使して効率良くコーディングできる一方、バッチアプリケーションに関しては ISPF や vi エディター上でコマンドを駆使してソースをメンテナンスする現場が今でも多くあるようです。開発手法についてもウォーターフォール型が主流で市場の変化に即納できないという問題も抱えているケースも少なくないと推測されます。このように開発ツールのみならずメソドロジーまでもが SoE と隔たりがあると、SoE 側のスキルを習得した若手技術者をバッチアプリケーション開発で起用することが難しくなり、将来的な技術者不足の懸念にまで発展する可能性もあります。

それでは、バッチアプリケーションをコスト効率の高い最新のデータセンター等へ移し、最新の開発環境で保守するにはどのような選択肢があるでしょうか。本書では、下表の3つの方式をピックアップして紹介します。

移行方式	利用製品	アプリケーションのコード	特徴
JCL を含めたリホスト	Micro Focus Enterprise Developer Micro Focus Enterprise Server	Native	IBM メインフレームからの移行であれば、既存資産への影響が最小
COBOL のみを対象としたリホスト	Micro Focus Visual COBOL Micro Focus COBOL Server	Native	> 移行元の環境は問わない > 実績が最も豊富
Java EE(jBatch) アプリとしてモダナイズ	Micro Focus Visual COBOL Micro Focus COBOL Server Java EE 7 対応 App サーバー	Java bytecode	> 移行元の環境は問わない > COBOL のビジネスロジックを維持しつつ Java EE で標準化されたフレームワークのメリットを享受

いずれの方式も最新のプラットフォームへの移行を可能とし、IDE 上で COBOL の専用のインテリセンスを用いたメンテナンスを可能とする点は共通です。本書では詳述を控えますが、いずれもエンタープライズアジャイルによる開発を見越した開発支援機能も利用できます [1]。

既存のアプリケーション並びに運用手順への影響を最小限に留めつつもプラットフォーム及び開発環境の刷新を目指すのであれば「JCL を含めたリホスト」方式が最適です。ただし、この方式に関しては IBM メインフレームからの移行が前提となります。移行元環境が IBM メインフレームでなければ「COBOL のみを対象としたリホスト」を選択する必要があります。プラットフォームや開発環境の刷新のみならず、Java EE 技術との連携、Java 資産との融合、JVM 向けのツールの利用、Java 技術者の人材活用等も見越したモダナイゼーションを目指すのであれば「Java EE(jBatch) アプリとしてモダナイズ」する方式が最適です。

本書では次章以降、これら各方式による移行技術について解説します。

1 エンタープライズアジャイル開発を見越した開発支援機能についてはデモ動画
「Enterprise Agile 型開発による COBOL アプリ開発」https://vimeo.com/213793228
もしくは White Paper「Micro Focus Visual COBOL を活用した開発作業の効率化」
http://www.microfocus.co.jp/resources/whitepaper/pdf/white_paper_079.pdf
を参照してください。

3 JCL を含めたリホスト

3-1 JCL を含めたリホストによるモダナイゼーションで期待できる効果

Micro Focus Enterprise Developer とその実行環境製品である Micro Focus Enterprise Server には JES をエミュレーションする機能を装備するアプリケーションサーバーが付属されます。本製品には JES のエミュレーションのみならず、CICS, IMS DB/DC, PL/I ランタイム機能も含まれます。本製品は最新の Windows OS や Linux/UNIX OS での動作をサポートしているため、これらのプラットフォーム上で JCL サブミットが可能です。サポート対象が OS であるため、仮想マシン上での運用も可能です。つまり、オンプレミスのみならずクラウドのようなスケーラブル環境への移行も可能であり、パフォーマンスや TCO の問題解決の糸口と成り得ます。

更に SoE の開発で利用されるような高機能なインテリセンスやビルド機能が装備された IDE を使って、COBOL, JCL, PL/I ソースのメンテナンスが可能です。更にこれらの開発ツールには Jenkins をはじめとした CI ツールとの連携を見込んだ自動ビルド機能、静的解析機能、単体テストユーティリティー等が付属しています。市場の変化に即応可能なアジャイル開発では、構成管理ツールとこのような CI 環境を連携させて可能な限りタスクをオートメーションさせることがイテレーションを進める上で重要となります。つまり COBOL の開発にて一般的なプラクティスに従ったアジャイル開発を導入することは不可能ではありません。

Micro Focus 社は 1976 年より COBOL コンパイラーの開発に従事してきました。本ソリューションで利用する製品 Micro Focus Enterprise Developer / Micro Focus Enterprise Server には Micro Focus 社が 40 年以上開発を重ねてきたコンパイラーが装備されます。このコンパイラーには IBM メインフレームとの言語互換、動作互換を持たせるオプション、EBCDIC データを扱うためのオプションをはじめ数百にも及ぶオプションが提供されています。

この Micro Focus Enterprise Developer / Micro Focus Enterprise Server を活用した「JCL を含めたリホスト」では以下のような効果が期待できます。

- 最新 OS への移行
  →より保守費用が低く、ハイパフォーマンスな環境で TCO を削減

- 既存データの継続利用
  → SAM, ISAM ファイルの継続利用が可能
  → EBCDIC データの継続利用が可能 [2]
  → IMS DB の継続利用が可能

- 既存の COBOL 資産の継続利用
  → IBM メインフレーム方言が入ったソースの解釈も可能
  →演算の中間結果の精度等言語仕様にないふるまいについてもメインフレームと合わせるオプションの利用が可能

- 既存の JCL 資産の継続利用
  → IBM JES2, JES3, VSE 方言をサポート [3]
  → IBM 標準のユーティリティーの大半をエミュレーションしているため、これらのユーティリティーを利用したステップの継続利用が可能

- 既存の PL/I 資産の継続利用

- その他 IBM メインフレーム資産の継続利用
  → CICS 連携、IMS DC 連携する資産の継続利用

2 オープン環境における標準の文字コードは ASCII コードです。他システムとのデータ授受等、汎用性を考慮し、移行のタイミングで ASCII コードへの変換を検討されるケースは多々みられます。この際、EBCDIC のソート順を前提としたプログラムロジックやゾーン十進項目の符号コードの差異を理由に断念されがちですが、Micro Focus は ASCII でデータを符号化しつつも照合順序を EBCDIC に合わせるオプションやゾーン十進項目の EBCDIC と共通の符号コードを持たせるオプションを用意しています。これらではカバーできない特別な理由がある場合に EBCDIC データの継続利用を検討します。

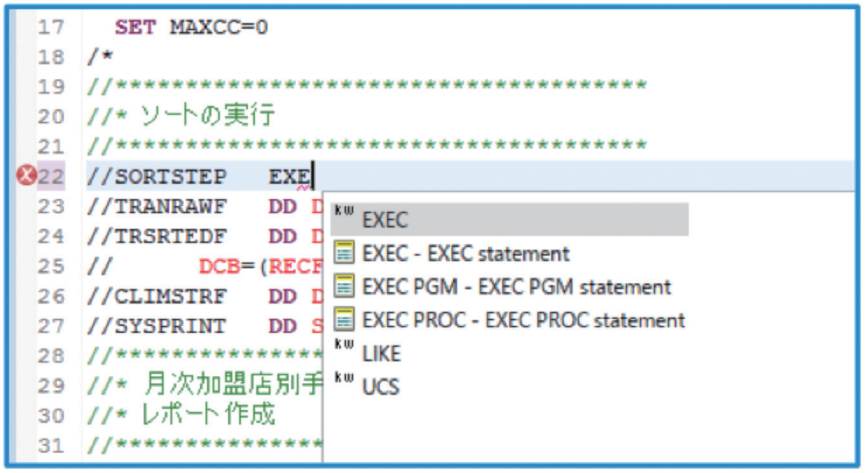
3 国産メインフレームで利用される JCL の多くは JES2 と文法上はほぼ共通であると言われています。そのため、国産メインフレーム固有のユーティリティーを利用しないステップで構成される JCL であれば本ソリューションを用いて移行できる可能性もあります。

> 最新の開発ツールの利用
→インテリセンス等が搭載された COBOL や JCL 専用の高機能なエディターを利用した効率的な開発
→ COBOL 専用のデバッガーを利用した効率的なデバッギング
→ IDE 上で埋め込み SQL 文が入ったソースの直接編集並びにデバッグを可能とする統合化プリプロセッサー機能の利用が可能
→ CI ツールとの連携を意識して設計された各種ユーティリティーを利用した各種作業の自動化

3-2 JCL を含めたリホストの開発

Micro Focus Enterprise Developer はこの「JCL を含めたリホスト」によるモダナイゼーションの開発作業を支援する製品です。本製品には対応するランタイム製品 Micro Focus Enterprise Server のテスト実行用のコンポーネントも含まれるため、テストフェーズまでご利用いただけます。本製品では Micro Focus 社が作りこんだ COBOL 専用の高機能なインテリセンスが搭載された COBOL エディターを Eclipse IDE や Visual Studio IDE で利用できます [4]。また、COBOL のみならず JCL 専用のエディターも用意されており、JCL の各種命令をオートコンプリートによる高速且つ正確なコーディング、バックグラウンドパーシングによる誤入力の即時チェック等が可能です。操作性に関しても Eclipse であれば Java のエディターと、Visual Studio であれば C# や VB.NET のエディターと同じ感覚で操作できるよう設計されており、Java や C#, VB.NET の開発者がこれらのソースをメンテナンスする感覚で COBOL や JCL のメンテナンスを行えます。

図3- 1 JCL エディターにおけるオートコンプリートのイメージ



3-3 アロケーションとカタログ

JCL ではステップ中の COBOL プログラム等が利用するデータ資源を DD 文で定義します。DD 文ではプログラムが参照する DD 名とデータセット名を DSN で紐付けます。DSN はデータの実体を表現した名称ではありますが、実際に記録 / 読み出しされる媒体についてはカタログにより決められます。このカタログに関する情報が DD 文中に記載されていない場合、メインフレームではユーティリティー等を利用し、カタログを行います。Micro Focus Enterprise Developer / Micro Focus Enterprise Server はオープン環境におけるファイルシステムに対してデータセットをカタログする機能が用意されています。本機能を利用することでオープン環境においても DD 文を維持することができます。

4 ViCOBOL エディターの実際のイメージについてはデモ動画
「進化する COBOL エディター ～ Eclipse 編～」https://vimeo.com/160987699
「進化する COBOL エディター ～ Visual Studio 編～」https://vimeo.com/160988056
等をご参照ください。

図3- 2 Administration Console 画面よりデータセットをカタログ

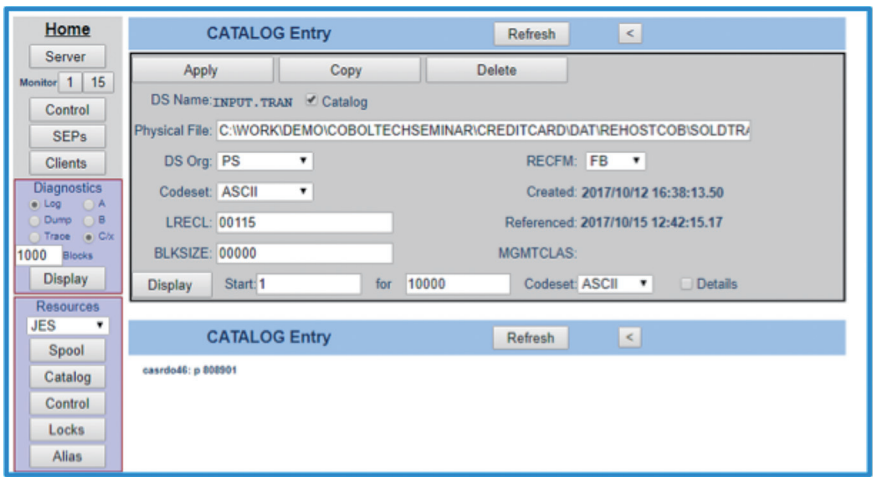
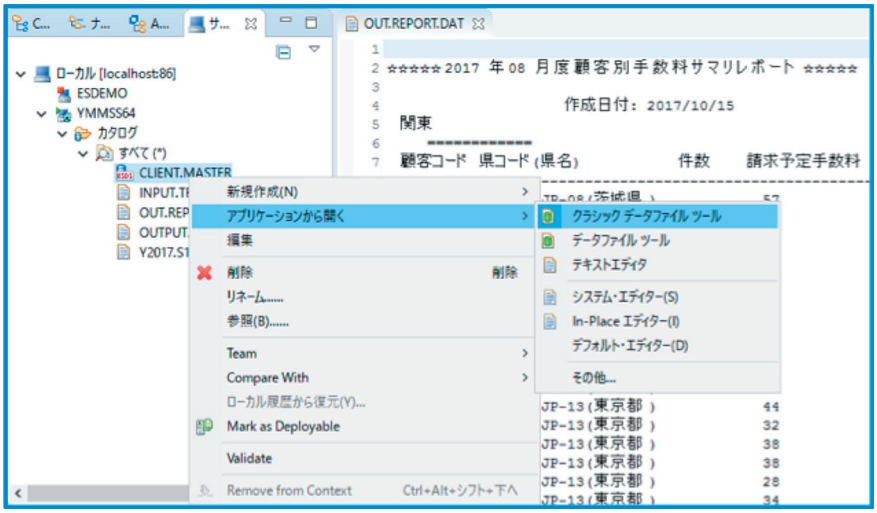


図3- 3 Eclipse IDE 中でカタログされたデータセットを参照



3-4 JCL を含めたリホストの運用

Micro Focus Enterprise Developer によりメンテナンスされた JCL についてはランタイム製品 Micro Focus Enterprise Server が導入された環境にて以下のような方法でサブミットできます。

- cassub コマンドラインユーティリティーによる JCL サブミッション
- Internal Reader へのコピーによる JCL サブミッション
- CICS アプリケーションから JCL をサブミット
- Enterprise Server Administration Console 画面より手動でサブミット

自動で運用しようと考えた場合、cassub コマンドラインインターフェイスの利用が想定されますが、本コマンドユーティリティーはあくまでもジョブのサブミットを処理するコマンドです。つまり、ジョブをサブミット後、結果を確認するには Administration Console 画面をチェックする等の対応が必要となります。スケジューラー機能を持った一般的な運用管理ツール製品と連動させるにはこの部分も自動化させる必要があります。こういった目的による利用も支援すべく、Micro Focus Enterprise Server には cassub を内包した MFBSI(Micro Focus Batch Scheduler Integration) という機能が付属されています。本機能を経由して cassub によるジョブサブミッションが行われると、MFBSI 機能がジョブ実行をモニターし、結果を戻り値として返します。これにより Micro Focus 製品を意識して開発をしていない運用管理ツール製品による自動運用が可能になります。

4 COBOL のみを対象としたリホスト

4-1 COBOL のみを対象としたリホストによるモダナイゼーションで期待できる効果

本ソリューションは本書で紹介するソリューションのうち Micro Focus 社がこれまで最も多くのお客様に対して提供してきたソリューションとなります。前章で紹介したソリューションは COBOL に加えて IBM メインフレーム上で稼動する JCL 等のメインフレーム独自資産の移行も前提としていました。しかし、本ソリューションにおける移行対象はプラットフォーム依存しない記法が可能な COBOL プログラム資産のみとなります。そのため、本ソリューションにおいては移行元環境に関する制約は特に設けていません。IBM のみならず国産メインフレーム、オフコン、オープンレガシーといった環境からの移行実績が多々あります。

本ソリューションによるモダナイゼーションの実現に必要な製品 Micro Focus Visual COBOL / Micro Focus COBOL Server は前章で確認した製品 Micro Focus Enterprise Developer / Micro Focus Enterprise Server にそれぞれ包含される製品です。前章で紹介した期待効果の多くが本ソリューションでも期待できます。

> 最新 OS への移行
→より保守費用が低く、ハイパフォーマンスな環境で TCO を削減

> 既存データの継続利用
→ SAM, ISAM ファイルの継続利用が可能
→ EBCDIC データの継続利用が可能

> 既存の COBOL 資産の継続利用
→ IBM メインフレーム方言が入ったソースの解釈も可能
→演算の中間結果の精度等言語仕様にないふるまいについてもメインフレームと合わせるオプションの利用が可能

> 最新の開発ツールの利用
→インテリセンス等が搭載された COBOL 専用の高機能なエディターを利用した効率的な開発
→ COBOL 専用のデバッガーを利用した効率的なデバッギング
→ IDE 上で埋め込み SQL 文が入ったソースの直接編集並びにデバッグを可能とする統合化プリプロセッサー機能の利用が可能
→ CI ツールとの連携を意識して設計された各種ユーティリティーを利用した各種作業の自動化

4-2 COBOL のみを対象としたリホストの開発

開発時に利用する製品 Micro Focus Visual COBOL は Micro Focus Enterprise Developer と同様に Eclipse IDE や Visual Studio IDE 上で COBOL の資産をメンテナンスできます。Micro Focus 社が COBOL 開発向けに独自に作りこんだ以下のような機能が反映された高機能な COBOL 専用のエディターが利用できます。

> COBOL 予約語のオートコンプリート

> COBOL のシンタックスに合わせたオートコンプリート
（例えば「MOVE ＜送り側データ＞」と記載すると MOVE 文のシンタックスに合わせて「TO」を自動で挿入します。）

> ユーザー定義語のオートコンプリート

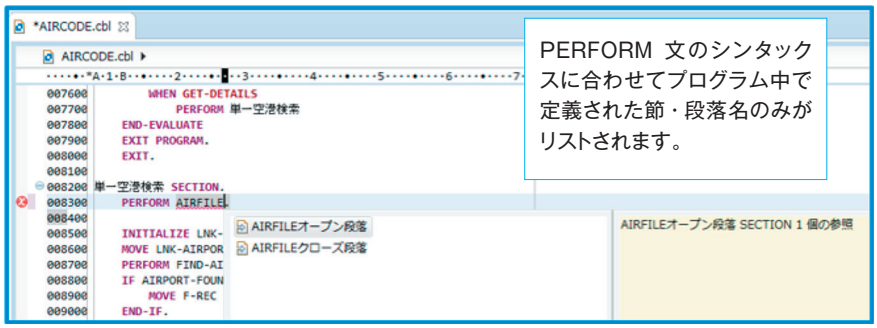
> コードスニペットの挿入

> 変数や節名等のリファクタリング

> 任意のブロックをコピーファイルとして自動抽出

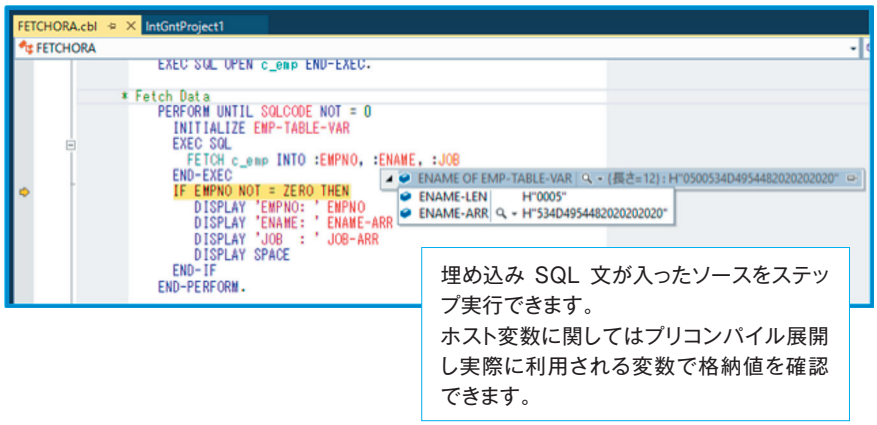
- 登録パターンに基づいた誤入力のオートコレクト
- 一連番号領域のリナンバリング
- 変数の定義位置と利用箇所のクロス参照
- コメントの切り替え

図4- 1 COBOL エディターにおけるオートコンプリートイメージ



上記のようにしてメンテナンスされた COBOL 資産は COBOL 専用のデバッガーを使ってステップ実行したり、任意の時点における変数の格納値をチェックすることが可能です。 Micro Focus Visual COBOL は Oracle Pro*COBOL 等のプリコンパラーの利用を前提としたソースについてもプリコンパル展開前のソースを直接メンテナンス・デバッギングする機能も付属されています [5]。これにより DB 連携プログラムの開発生産性が一層向上します。

図4- 2 埋め込み SQL 文が入ったソースのデバッギング



5 Pro*COBOL の連携をはじめ、DB2、ODBC・ADO.NET・JDBC を経由したアクセスを可能とする Micro Focus が独自に用意する統合化プリプロセッサー OpenESQL がこの目的で利用可能です。これらの機能の具体的な利用イメージについては下記 White Paper 等をご参照ください。
「COBOL アプリケーションのデータアクセス」
https://www.microfocus.co.jp/resources/whitepaper/pdf/white_paper_072.pdf

Micro Focus Visual COBOL にはエディターやデバッガーのみならず、以下のような様々なユーティリティーも付属しています。これらはいずれも自動実行が可能であり、Jenkins をはじめとした CI ツールとの連携も想定して機能設計されているものもあります。エンタープライズアジャイル型の開発を見込んで CI 環境上における各種工程の自動化に取り組むことも可能です [6]。

- Ant や MSBuild による Eclipse IDE 及び Visual Studio IDE で開発したプロジェクトベースのビルド
- COBOL 専用の単体テスト
- COBOL 専用の静的コード解析 [7]
- COBOL 専用のコードカバレッジ
- COBOL 専用のパフォーマンス解析

6 本コンセプト及び自動化取り組みへのイメージについては下記デモ動画をご参照ください。
「Enterprise Agile 型開発による COBOL アプリ開発 前編」
https://vimeo.com/213793228

7 Visual COBOL にビルドインされている解析ルール以外のルールを実行させたい場合は、Micro Focus Enterprise Analyzer でルールをカスタマイズ作成することが可能です。同製品で自作したルールは Visual COBOL にインポートして利用することができます。

4-3 シェルや Windows Powershell を利用したジョブの実行

本ソリューションにおいては COBOL を除いたメインフレーム資産は再利用しません。そのため、移行元がメインフレームの場合、移行先環境に対応するエミュレーション機能がないものは移行先環境で利用できる機能で代替します。下図は各要素の典型的な移行･代替イメージとなります。

図4- 3 典型的な移行・代替イメージ

ジョブコントロールに関しては移行先の OS 上で利用可能なシェル言語を利用して記述します。そのため、作成したシェルファイルについては Micro Focus 製品を意識して開発されていない、ターゲット OS 向けの一般的な運用管理ツールからスケジュールに基づいた起動が可能です。アプリケーション資産に関しては、ターゲット OS 向けの実行形式モジュールや共有ライブラリ等へビルドすることが可能です。Micro Focus オリジナル形式の動的ロードモジュールへビルドするオプションも用意されています。この形式のモジュールは、実行形式で提供されるラウンチャーから起動します。いずれの形式にビルドしましても COBOL に関して特別な考慮をすることなくシェル中に起動命令を記述できます。新環境においては論理ファイルと物理ファイルの関連付けを環境変数に指定します。ユーティリティーに関してはターゲット OS のファイルシステムに関連するものは OS コマンドで代替します。ソートやファイル編成変換等、COBOL のデータファイル固有の処理を必要とするものは Micro Focus Visual COBOL に付属するユーティリティー MFSORT や REBUILD 等で既存の処理を引き継がせます。これらのユーティリティーも各 OS の実行形式として提供されているため、特別な考慮をすることなくシェルから起動命令を発行することが可能です。

5 jBatch(JSR352) とは

5-1 JVM 向けフレームワークの選択肢

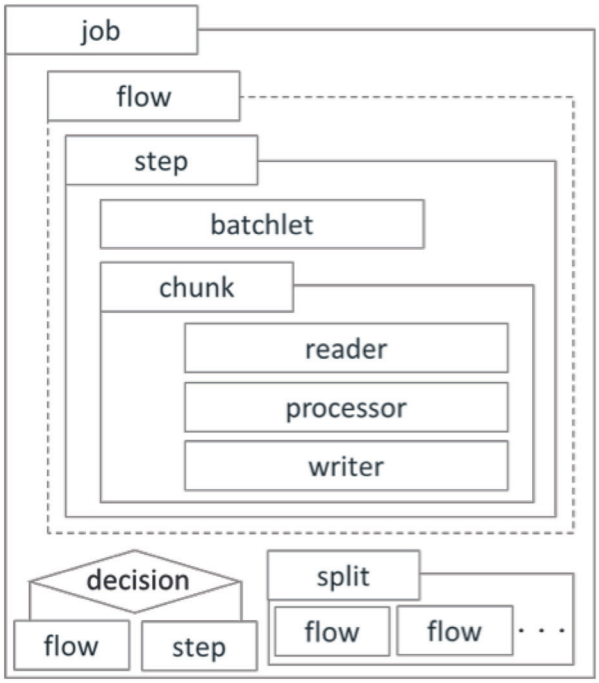
JVM(Java virtual machine) は、Java bytecode として定義された命令セットを実行するスタック型の仮想マシンです。この Java bytecode を生成する JVM 言語と称される言語は Java をはじめ 50 以上にも及びます。Micro Focus COBOL のように既存の言語を JVM 言語として実装した言語も多数あります [8]。JVM 向けの技術について論じる場合、これら全ての JVM 言語に関連する全てのコミュニティーや団体から提供される技術が対象となり、非常に著大なコミュニティーで構成される技術を指します。Java 言語だけでも 58 万人から構成される巨大なコミュニティーが活発的に活動しているようです [9]。巨大なコミュニティーが活発的に活動するということは、多様な技術が多くのフィードバックを受けつつ頻繁に改善されていることを示唆しています。

その中でバッチアプリケーション向けに応用可能な技術としては jBatch, Hadoop, Spark 等が挙げられます。Hadoop 及び Spark はビッグデータの分野で主に利用される並列分散処理の代表的なフレームワークです。COBOL を JVM 言語として扱うことで既存のビジネスロジックを同フレームワーク上で再利用し、アプリケーションのパフォーマンス向上を図ることが可能です [10]。jBatch は「Batch Application for Java Platform」という公式名を持つ JSR 352 として JCP(Java Community Process) Executive Committee により正式に承認された Java EE の公式フレームワークです。つまり JVM におけるエンタープライズ向け技術で最も巨大なコミュニティーによりサポートされるフレームワークとなります。本書では数ある JVM 向け技術のうち本フレームワークにフォーカスし、COBOL のバッチアプリケーションを JVM アプリケーションとしてモダナイズする技法について論じます。

5-2 jBatch の基本概念

jBatch ではジョブやステップを JSL(Job Specification Language) と呼ばれる XML ファイルで規定します。メインフレームにおける JCL に相当する役割を担います。JSL ではジョブやステップに加えて、JCL にはない様々な要素を指定できます。下図は JSL で指定可能な主な要素の関係を図示化したものになります。

図5- 1 JSL の主な構成要素



8 参考「List of JVM languages」
https://en.wikipedia.org/wiki/List_of_JVM_languages (リンク確認 : 2018/02/14)

9 参考「Why Learn Java?」
http://www.bestprogramminglanguagefor.me/why-learn-java (リンク確認 : 2018/02/14)

10 Hadoop/Spark 上で COBOL を利用する技法については下記 White Paper をご参照ください。
「Micro Focus Visual COBOL 2.3J for x64/x86 Linux Hadoop/Spark 連携アプリケーション (Oracle Big Data Appliance、Asakusa Framework 利用) 動作検証結果報告書」
http://www.microfocus.co.jp/resources/whitepaper/pdf/white_paper_082.pdf

各要素の概要を下表に記します。

要素名	概要
job	バッチ処理全体を包括するエンティティです。JSL ファイルにおいては第 1 階層に記述します。実行時は job 要素の id 属性に指定した名前を使ってインスタンスを起動します。
step	ジョブステップの定義をします。子要素には batchlet、もしくは chunk を指定しステップの種類を規定します。 起動回数に制限を設ける start-limit 属性や完了したステップを restart 時に再度実行させる allow-start-if-complete 属性の指定も可能です。続くステップや flow がある場合は next 属性にそれらの id を指定します。
flow	flow は子要素として指定される step, flow, decision, split をまとめて 1 つの論理的なエンティティとして扱います。flow に続くステップや flow がある場合はそれらの id を next 属性に指定します。
batchlet	batchlet は親要素である step 属性が batchlet というタスク指向の方式で処理されることを示します。
chunk	batchlet は親要素である step 属性が chunk 指向[11]の方式で処理されることを示します。Chunk 方式では構成に基づき checkpoint が処理されます。子要素に は reader, processor, writer, skippable-exception-classes, retryable-exception-classes が指定できます。属性には checkpoint-policy、item-count、time-limit、skip-limit、retry-limit が指定可能です。
decision	decision は ref 属性で指定する javax.batch.api.Decider インターフェイスを実装するクラスを使って step, flow, split の順番をカスタマイズするための要素です。
split	split は子要素として指定する flow を並列同時実行させるための要素です。この場合、各 flow はそれぞれ独立したスレッドで実行されます。子要素である全 flow が終了した時点で split は完了となり next 属性で指定した要素に処理が遷移します。

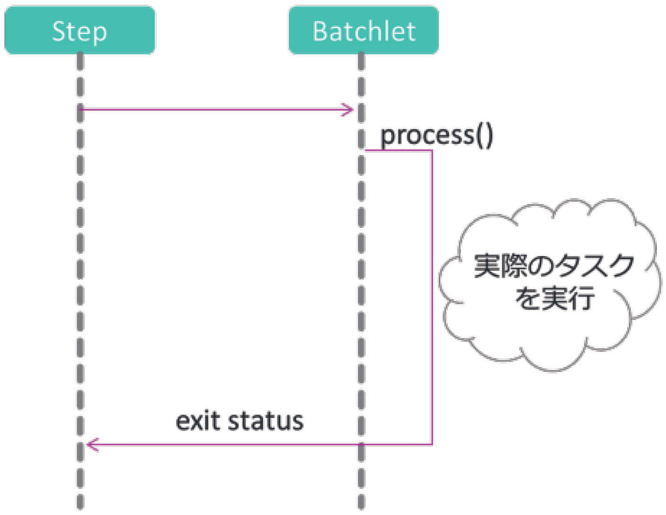
JSL では Job, Step, Chunk, ItemReader, ItemProcessor, ItemWriter の開始や終了時にインターセプトさせたい処理をリスナーとして登録ができます。 例えば Job のリスナーであれば javax.batch.api.listener.JobListener インターフェイスを継承した batch artifact クラスを用意し、それを job の子要素として指定します。次々項で紹介する Skip 例外や Retry 例外についても実際に Skip される際、Retry される際にリスナーとして処理を挿入させることが可能です。

5-3 jBatch のステップモデル : Batchlet

Batchlet はタスク指向のモデル、もしくは unmanaged タイプのモデルとカテゴライズされます。こちらのタイプについては Step 内でどのような処理をすべきか規定されていません。Batchlet Step が起動されると javax.batch.api.Batchlet インターフェイスを実装した batch artifact における process() メソッドが呼ばれ、本メソッド内で必要なタスクを実行させます。もちろんこのメソッドから別のメソッドや機能を利用しても構いません。コマンド実行、ファイル転送、ファイルのマージやソート等でも利用できます。

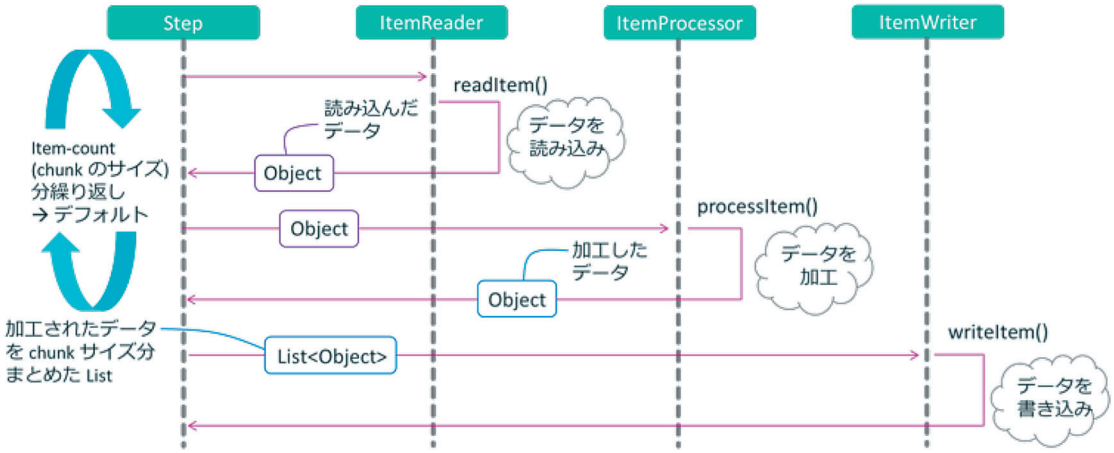
11 Chunk 指向とはデータの読み込みと加工の繰り返しにより生成される Chunk を同一のトランザクション境界内で書き出す方式を指します。Chunk 指向による処理内容の詳細については 5-4 項で紹介します。

図5- 2 Batchlet Step の処理フロー



5-4 jBatch のステップモデル : Chunk

Chunk は Batchlet と異なり managed タイプの Step とも言われます。Chunk ではデータを読み込み、加工し、出力するフローで構成されるタスクの処理に利用されます。処理中はバッチランタイムにより定期的にチェックポイントが行われます。このチェックポイントの仕組みにより高度なリカバリも可能となります。下図はこの Chunk タイプの Step の処理フローを抽象的に表現したものとなります。



以下は典型的な Chunk ステップの処理フローとなります :

① javax.batch.api,chunk.ItemReader インターフェイスを実装した Reader artifact の open(Serializable checkpoint) メソッドが呼ばれ、入力データが格納された RDB との接続を確立、もしくはファイルのオープンが行われます。

② Reader artifact の readItem() メソッドが呼ばれ、入力データが読み込まれます。

③ Reader artifact から戻されたデータを javax.batch.api,chunk.ItemProcessor インターフェイスを実装した Processor artifact の processItem(Object item) メソッドの引数に渡し、データをビジネスロジックに基づいて加工します。

④ Processor artifact により加工されたデータはいったん List に溜められます。

⑤ ②～④までの処理を Chunk 要素の item-count 属性に指定された数だけ繰り返されます [12]。

⑥ javax.batch.api,chunk.ItemWriter インターフェイスを実装した Writer artifact の open(Serializable checkpoint) メソッドが呼ばれ出力データを書き込む RDB との接続を確立、もしくはファイルのオープンが行われます。

⑦ Writer artifact の writeItem(List<Object> items) メソッドが呼ばれ、引数として渡された⑤で生成された List 中のデータの書き込みが行われます。

この①～⑦までで1つのチェックポイントとなります。このチェックポイント単位で処理されるアイテムが chunk と呼ばれています。つまり、デフォルトであれば 10 件分の処理アイテムが chunk であり、例えば 2000 件の入力データがあれば 200 回のチェックポイントが行われることになります。トランザクションについてはチェックポイント毎に分離されます。そのため、例えば 8 回チェックポイントを行った後に例外等により処理が中断されても最初からやり直すことなく、トランザクション確定済みの 8 回目のチェックポイント以降より処理をリカバリすることが可能になります [13]。

Chunk のステップ処理中に Reader, Processor, Writer といった artifact 内で例外が発生するとバッチ処理は FAILED ステータスで終了します。しかし、Skip 例外もしくは Retry 例外を chunk 要素の子に指定することでこの挙動をオーバーライドすることも可能です。Skip 例外として指定された例外は、処理中に投げられてもランタイムは処理を止めずに処理を継続します。スキップする回数はデフォルトは無制限ですが、chunk 要素の skip-limit 属性を構成することでスキップ許容回数の上限を設けることも可能です。Retry 例外は指定した例外が発生した場合に、例外が発生した未コミット状態のチェックポイントをリトライさせる機能です [14]。Skip 例外同様、リトライ回数はデフォルトでは無制限ですが chunk 要素の retry-limit 属性を構成することで上限を設けられます。両者ともに対象の例外のスーパークラスを Skip/Retry 例外として指定し、Skip/Retry 対象外とする例外を exclude class として指定することも可能です。

5-5 jBatch の運用方法

バッチジョブのオペレーションは javax.batch.operations.JobOperator インターフェイスの実装を経由して行います。本インターフェイスには下表のようなメソッドが定義されており、これらを用いてオペレーションします。

メソッド名	概要
start(String jobXMLName, Properties jobParameters)	ジョブインスタンスを生成し、生成した インスタンス上で最初のジョブ実行試行を行います。 引数に指定する jobXMLName はクラスローダーの META-INF/batch-jobs ディレクトリ配下にある JSL の名前です。
restart(long executionId, Properties restartParameters)	FAILED ステータスもしくはSTOPPED ステータスのジョブを再開させます。
stop(long executionId)	指定したジョブに停止命令を発行します。あくまでも停止命令の発行を担うメソッドであり、ジョブの作りによっては停止されない可能性もあります。
abandon(long executionId)	指定したジョブを ABANDONED ステータスへ変更します。ABANDONED ステータスのジョブは restart できません。

12 chunk 要素の checkpoint-policy 属性並びに item-count 属性のデフォルトはそれぞれ「item」、「10」であり、10 個分の要素を持つ List が生成されます。この数については item-count 属性で構成可能です。time-limit 属性も併せて指定している場合は、item-count 分 List が溜まっていなくとも本属性に指定した秒数が経過すると⑥の処理が開始されます。checkpoint-policy 属性に「custom」を指定する場合は、checkpoint-algorithm 要素も併せて指定します。この場合、⑥の処理は同要素に指定した artifact 中のロジックに基づいて開始されます。

13 step 要素の allow-start-if-complete 属性が true（デフォルトは false）の場合はステップの初めから再開されます。

14 デフォルトでは例外が発生した時点で処理中の chunk を Rollback し、item count をリセットした上で対象の chunk を初めから再実行します。Rollback させずに処理中の chunk の item count からリトライさせることも可能です。

JobOperator インターフェイスの実装については jBatch を実装する Java EE アプリケーションサーバーにより提供されます。JobOperator のインスタンスは javax.batch.runtime.BatchRuntime クラスの getJobOperator() メソッドより取得が可能です。このインスタンスを取得することで様々な Java プログラムからオペレーションが可能になりますが、現実的な運用方法としては以下が挙げられます。

> RESTful Web サービスとして起動
JAX-RS で開発する RESTful Web サービス中に上記ロジックを埋め込み、任意の URI がたたかれるとジョブが起動されるように構成します。

> Web アプリケーション経由で起動
JSP/Servlet や JSF 等に上記ロジックを埋め込み、特定のボタンが押下されるタイミング等でジョブが起動されるように構成します。

> EJB タイマーによる自動起動

また、多くの Java EE アプリケーションサーバーにはコマンドラインユーティリティーが装備されており、JobOperator インスタンスの取得並びに任意のオペレーションメソッドの実行を内部的に行うコマンドも用意されています。この機能を利用して運用管理ツール等から自動起動させることも可能です。

6 Java EE(jBatch) アプリとしてモダナイズ

6-1 JVM COBOL とは

Micro Focus が提供するコンパイラーは、ターゲット OS の Native コードへコンパイルする機能の他に Java bytecode や .NET の MSIL コードへコンパイルする機能も装備しています。生成するコードの切り替えはコンパイラーオプションで操作が可能です。Java と同様に Java bytecode へコンパイルする場合は、JVMGEN コンパイラー指令を指定します。このように Java bytecode へコンパイルし、COBOL プログラムから JVM クラスを生成する機能が JVM COBOL です。JVM COBOL は長年の実績がある Micro Focus のコンパイラーの基本部分を引き継いでおり、サポートする COBOL シンタックス並びにコンパイラーオプションの大半が共通です。国際規格に準拠した手続き型の COBOL プログラムはもちろんのこと IBM メインフレーム方言が含まれた COBOL プログラムをも JVM クラスとしてコンパイルすることが可能です。加えて JVM COBOL では Java で表現可能な命令の多くを COBOL でも表現可能にした拡張シンタックスも利用できます。例えば、COBOL をオブジェクト指向型で記述して Java で書かれたインターフェイスの実装や親クラスの継承を行うことが可能なため、Java の特徴の1つであるポリモーフィズムを COBOL で実現することができます。この他にも Annotation に相当する記述や TRY – CATCH による例外ハンドリング等もサポートされており、JVM アプリケーションとして移行後、JVM 環境と親和性の高いコーディングで機能強化・追加することも可能です。挙動に関するオプションについても引き継がれているため、ゾーン十進変数の符号部を EBCDIC として扱うことや、演算時の中間結果の精度を IBM メインフレームと合わせることも可能です。JVM COBOL を活用して JVM アプリケーションを構築することの利点を整理すると以下のようになります :

> 既存の COBOL 資産を活かしたまま JVM アプリケーション化が可能
→ Java をはじめとした他の JVM 言語への書き換えコストを回避
→正しい動作実績がある資産が継続利用できるため、品質劣化リスクを回避

> 既存のデータアクセス方式の維持が可能
→ Native 環境と同等のファイルハンドラーを JVM COBOL でも利用可能

> JVM COBOL 拡張方言を利用した機能強化・追加
→ 他の JVM 資産の呼び出し、実装、継承等が可能であり、Java で書かれたアプリケーション資産を COBOL から活用可能
→ JVM 向けのフレームワーク等と親和性の高いコーディング
→ Java と同様のロジックの組み上げが可能なため、COBOL に馴染みのない Java エンジニアが COBOL を習得する際の抵抗感を軽減

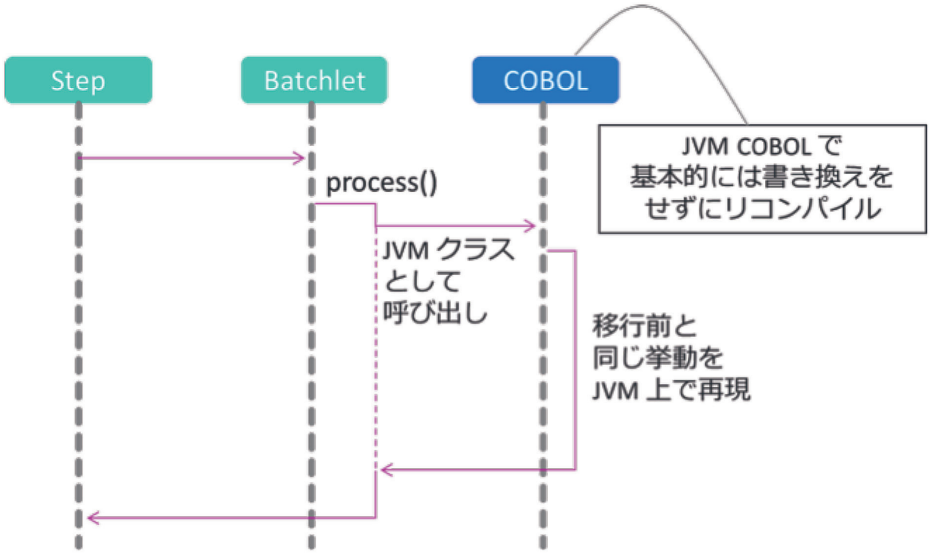
> COBOL エンジニアの継続活用
→ COBOL エンジニアと Java エンジニアがペアプログラミングのように連携して業務することによる技術移転→次世代の COBOL エンジニアの育成

このように JVM COBOL を利用することで既存の COBOL 資産を継続活用しながら JVM アプリケーションとして運用することのメリットを享受することが可能です。JVM におけるエンタープライズアプリケーション開発の標準ともいえる Java EE において前章で紹介した jBatch が Java EE 7 より Java EE のバッチアプリケーション開発の標準化フレームワークとして加わりました。JVM COBOL を利用した場合、既存の COBOL 資産をいかにしてこの jBatch に組み込めるか次項以降で見ていきます。

6-2 COBOL 資産の jBatch 化（Batchlet タイプのステップ）

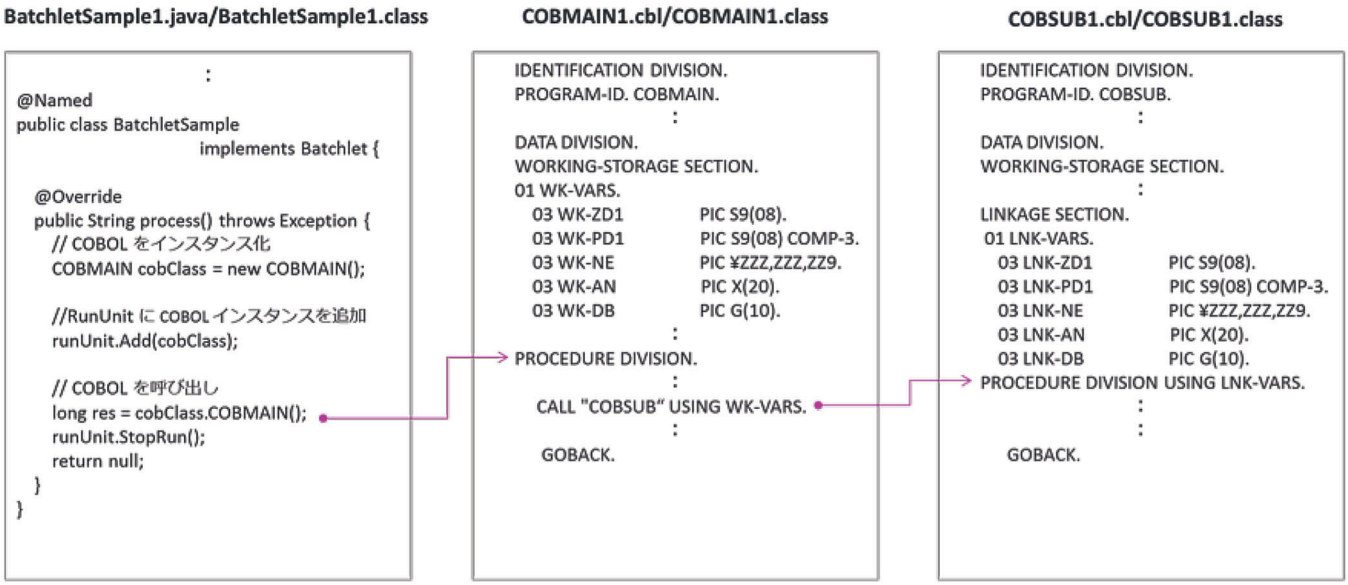
jBatch に関しては前章で紹介したように Batchlet 及び Chunk と2種類のステップタイプが用意されています。Batchlet に関しては unmanaged 形式のステップタイプとなり、Batchlet artifact 内で処理したいタスクを自由に記述できます。そのため、Batchlet のステップ中で COBOL 資産を利用する場合は Batchlet artifact の process メソッドより対象の COBOL モジュールを JVM クラスとして呼び出します。

図6- 1 Batchlet ステップ内で COBOL 資産を利用



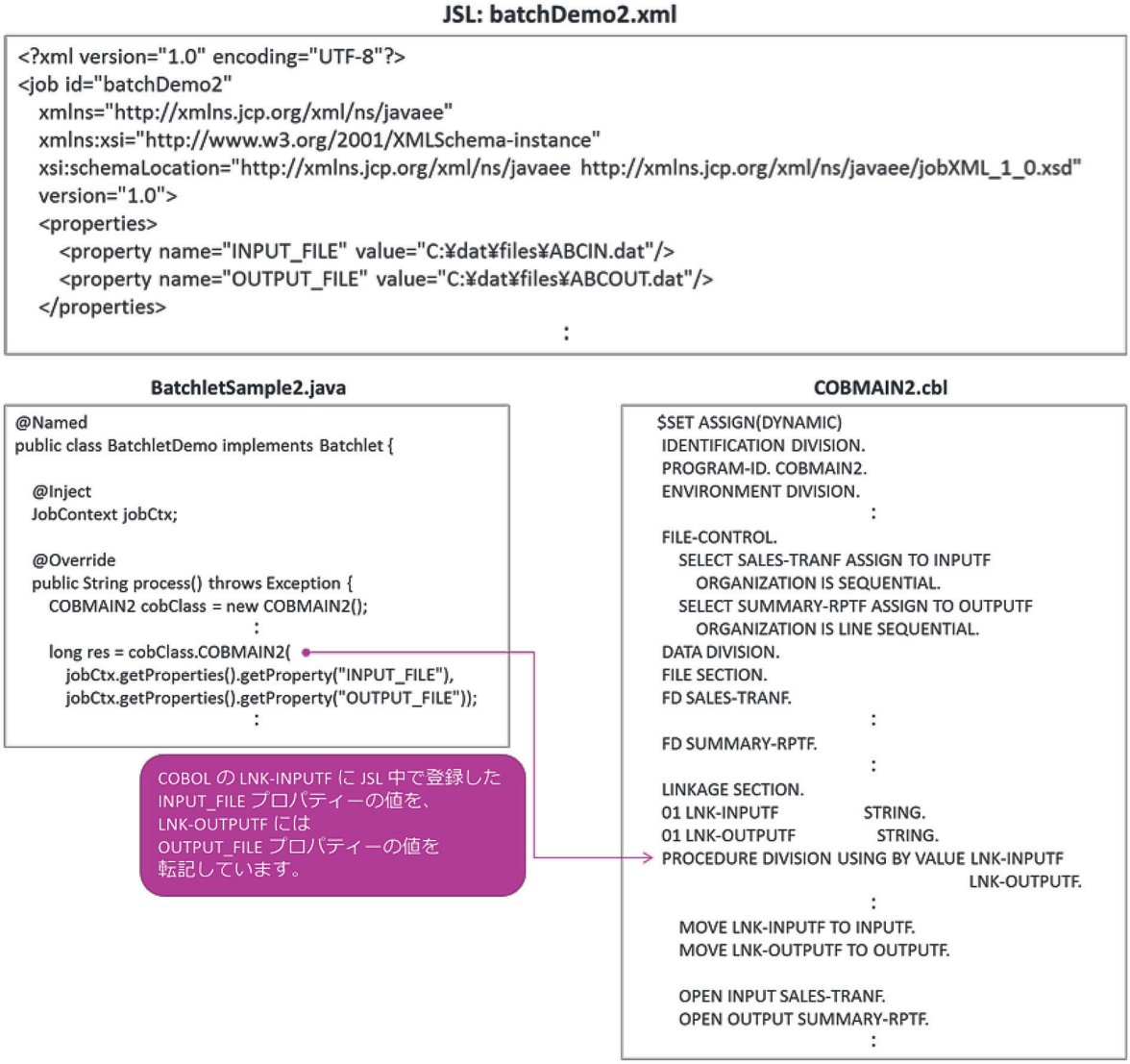
Batchlet artifact より直接呼び出すのは現行 JCL で運用している場合、EXEC 文で指定するプログラムです。既にオープン化済みでシェル等で運用している場合は、対象の実行形式等のメインプログラムです。対象の COBOL プログラムについては一般的な JVM クラスとして呼び出しが可能です。手続き型の COBOL プログラムが JVM クラスとしてコンパイルされると、プログラム中の命令はプログラム名と同名のインスタンスメソッドとして実装されます。DATA DIVISION の WORKING-STORAGE SECTION に定義された変数は Native 環境同様に静的なデータとして扱われます。そのため、インスタンス間で WORKING-STORAGE SECTION 中のデータを干渉させたくない場合は、Micro Focus COBOL Server が提供する com.microfocus.cobol.runtimeservices.RunUnit を利用します。本機能は RunUnit のインスタンス単位で境界を分けます。COBOL に処理が移りますと、.class の拡張子を持つ JVM クラスにコンパイルされていようとも COBOL プログラム同士であれば Native 環境と同様に CALL 文でサブモジュールの呼び出しが可能です。Batchlet artifact、COBOL メイン、COBOL サブまでの典型的な呼び出しフローイメージを図 6-2 に示します。

図6- 2 Batchlet による COBOL の呼び出しイメージ



このように Batchlet タイプであれば jBatch 化にあたり、このようにジョブステップを担う COBOL プログラムを一行も書き換えずとも artifact から Java で書かれた JVM クラスを呼び出すのと同じ感覚で COBOL を呼び出すことができます。JCL やシェル等からアプリケーションを実行する際は、JCL であれば DD 文或いはカタログで、シェル等であれば環境変数で論理ファイルと物理ファイルのパスを紐付けていました。jBatch にした場合も環境変数経由で紐付けは可能です。ただ、jBatch では JSL 上で構成する Job や Step 等の Property にて入出力に必要な接続情報等を管理することが一般的です。IO 処理の対象が COBOL のデータファイルであってももちろん　Property による管理が可能です。Micro Focus の COBOL コンパイラーは ASSIGN(DYNAMIC) 指令を指定してコンパイルした場合、File Control Entry の SELECT 句中の ASSIGN 句に続いて指定するファイル名と同名の変数が宣言されていなければ、その名前で PIC X(255) の変数を自動で生成します。OPEN 文が発行される前にこの変数にファイルパスを格納することで、同変数中のパスにあるファイルが参照されます。この機能を利用して、Batchlet の artifact より Property に登録されたファイルパスを COBOL に渡し、そのデータを ASSIGN(DYNAMIC) により自動生成される変数に転記すれば、JSL に構成したパスを COBOL のデータファイルとして利用することが可能です。図 6-3 にその記述例を記します。

図6- 3 Job プロパティーを利用したファイルの割り当て



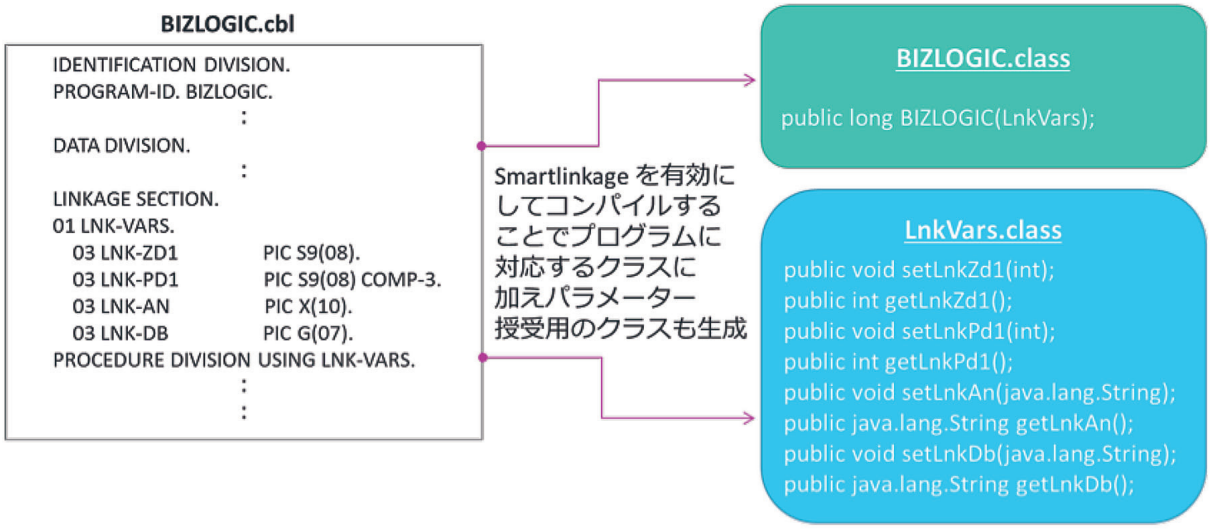
6-3 COBOL 資産の jBatch 化（Chunk タイプのステップ）

Chunk タイプのステップは、Reader, Processor, Writer により構成されます。Chunk ステップ化にあたっては対象の COBOL 資産に対して、まずデータ入力、データ加工、データ出力処理で仕分けします。多くの場合、このうちのデータ加工部分については企業独自の業務仕様に基づいてカスタマイズ作成されるビジネスロジックが入り込みます。このビジネスロジックについてはなるべく書き換えせずに Processor artifact から再利用する方法を検討します。ビジネスロジックとして利用されるプログラムはサブモジュールとして組み上げられることが多く、パラメーター授受が発生します。COBOL はゾーン十進、パック十進、固定長の英数字項目、SJIS 等を格納する2バイト文字項目等、Java の基本型にはないデータタイプが多く使われます。Java の数値の基本型はバイナリ形式で保持されるため、Java にて例えば int 型の変数に格納された数値をパック十進形式で受け取る COBOL に渡すには格納された数値を1バイト2桁で保持するパック十進に変換するモジュールを用意してワンクッションかませる必要があります。String に関しても内部では Unicode で値保持されますが、SJIS でデータ保持する PIC N(10) の引数に渡すのであれば、一度 SJIS に変換して、更に COBOL の宣言に合わせて10文字分のサイズに調整する必要があります。

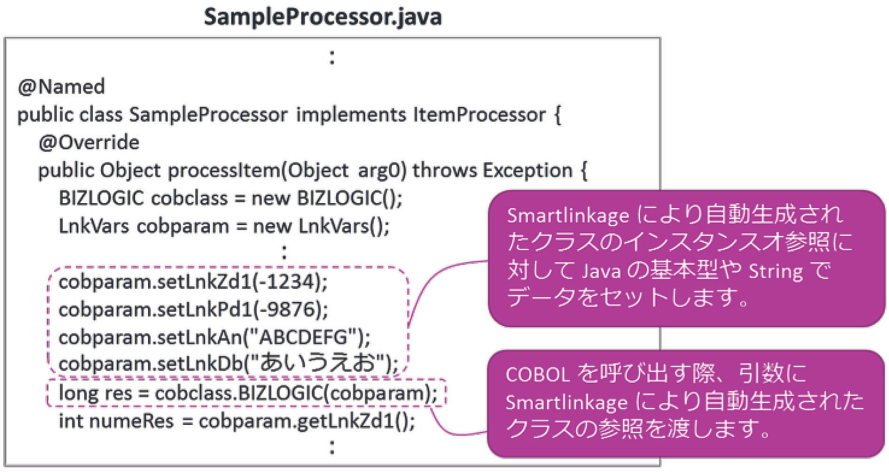
このように多くのエンジニアリングコストを伴う厄介なプログラム間のデータ型差異については、Micro Focus COBOL Server のランタイムが透過的に差異を吸収する Smartlinkage という仕組みを提供しています。Smartlinkage は、対象の COBOL プログラムにおける LINKAGE SECTION 中の 01 アイテム毎に Java Beans 形式のリソースを生成します。このリソースの setter/getter の引数、戻り値は Java の基本型や String で実装されます。Java の開発者はこの setter/getter を通じて値の授受を行います。このリソースのインスタンス参照を COBOL を呼び出す際の引数に指定すると Micro Focus のランタイムが自動的に LINKAGE SECTION 中で実際に宣言された型へ変換を行います。図 6-4 は Smartlinkage によるビジネスロジックプログラムの呼び出しイメージとなります。

図6- 4 Smartlinkage によるビジネスロジック開発イメージ

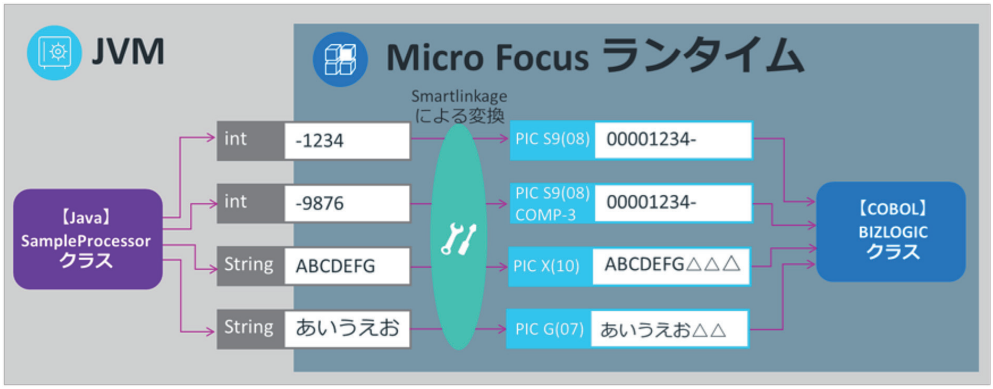
① Smartlinkage を有効にしてコンパイル



② コンパイルされた JVM クラスを Processor から利用

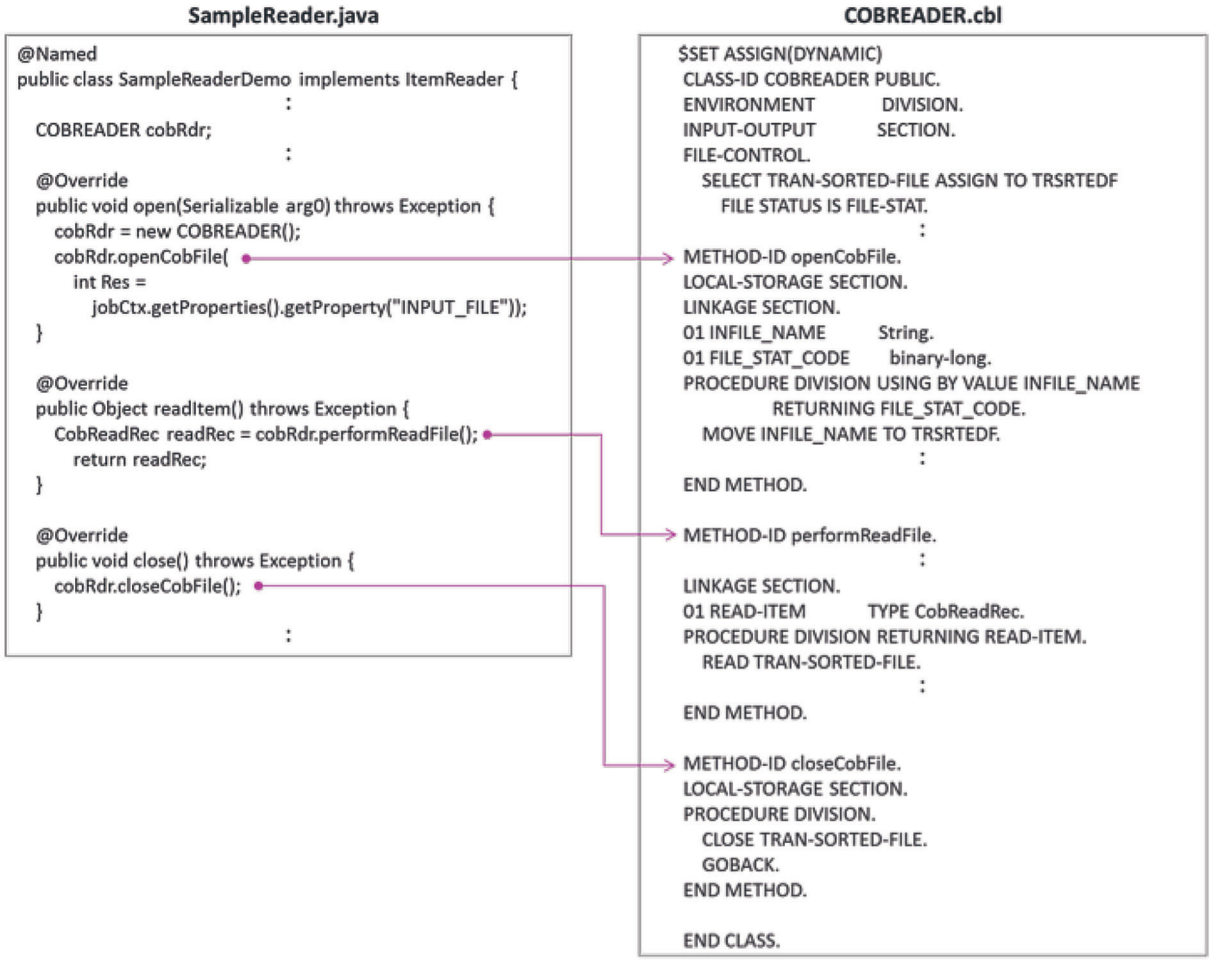


③ 実行



データ入力に関しては順ファイルや索引ファイルを使った運用を継続させる場合は、データファイルとの IO 部分は COBOL に処理させます。しかし、jBatch のランタイムに合わせて Reader artifact の open メソッドにより入力データが格納されたファイルをオープンし、readItem メソッドによりデータを読み込むよう構造を整合させる必要があります。ここは手続き型の COBOL で対応させることも可能ですが、下図の要領でオブジェクト指向型の COBOL で対応させることも可能です：

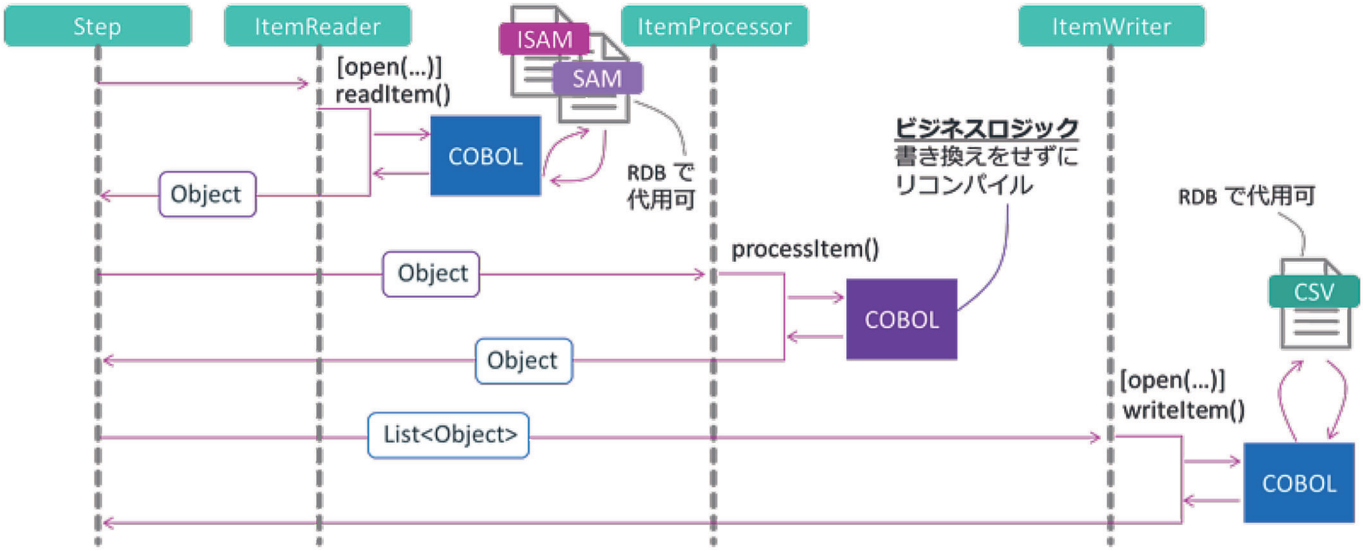
図6- 5 Reader による COBOL データファイルアクセスロジック例



データ出力に関しては、現行のような帳票データを出力することも可能ですが、より効果的にデータ活用させるべく BI ツール等で読み込みが可能な CSV や RDB に出力させるという選択肢もあります。いずれの形式にするにしてもこれらとのインターフェイスを COBOL で行うことも可能です。もちろん COBOL 固有のデータを利用する等特別な理由がなければ、Java で Writer を構築いただくことも可能です。

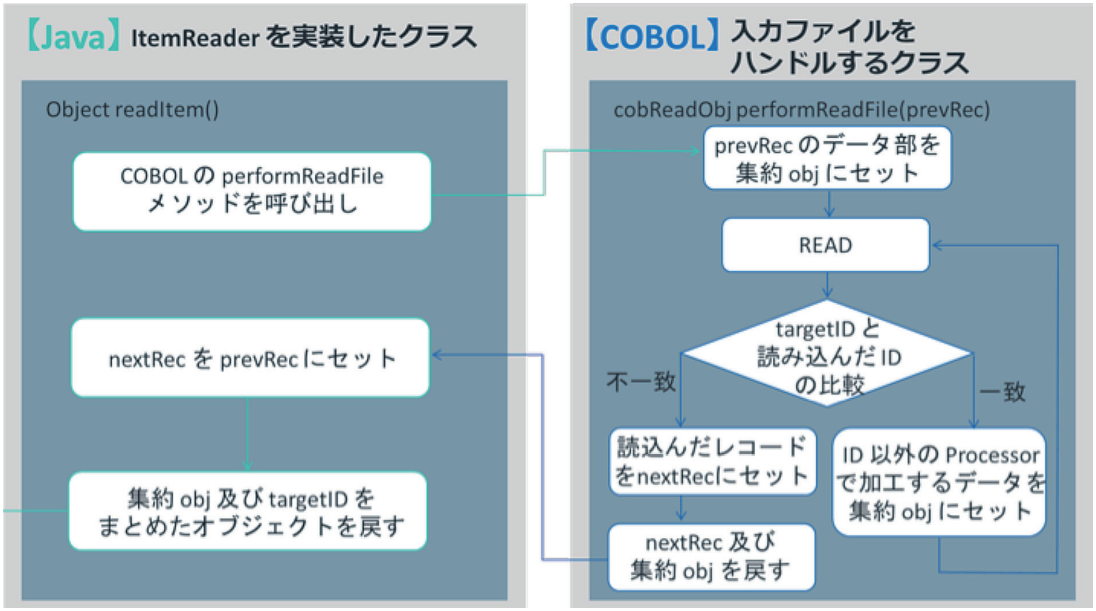
この全体の流れを抽象的なフローで図示化したものが下図となります：

図6- 6 Chunk タイプによる COBOL と Java の関係



上図のフローのように Chunk タイプによる設計は Reader à Processor à Writer の構造になっており、入力レコードと出力レコードの数が1：1の関係となるモデルになります。しかし、アプリケーションによっては、顧客や支店毎に集約して分析を行います。このような構造の設計のアプリケーションであっても工夫次第で Chunk 化は可能です。一例としては、まず Chunk ステップに入る前に順ファイルを目的のコードでソートをします。Chunk のステップでは Reader の1回の readItem メソッドの実行で目的のコードが同じ間は読み込みを繰り返し、集約した結果を readItem の戻り値に設定します。図 6-7 にこの概念に基づいたサンプルフローを示します：

図6- 7 入力レコードの集約化の例



ItemReader を実装したクラス			
readItem()			
戻り値			
readObj	targetID	集約する対象の ID(例：顧客 ID、支店 ID 等)	
	集約 object	Processor で加工する ID 以外のデータを集約したオブジェクト	
入力ファイルをハンドルするクラス			
performReadFile			
引数			
prevRec	targetID	集約する対象の ID(例：顧客 ID、支店 ID 等)	
	ID 以外のデータ部	入力レコード中の ID 以外のデータ	
戻り値			
cobReadObj	nextRec	prevRec の targetID と異なる ID を含む最後に READ して読込んだレコード	
	集約 object	Processor で加工する ID 以外のデータを集約したオブジェクト	

もしくは Chunk ステップに入る前にデータを集約するステップを挟んでも同等の結果は得られます。このように Chunk モデルへ適用するには多少の書き直しも伴いますが、Chunk の利点を鑑みますと検討の価値はあります。

6-4 COBOL 資産の jBatch 化における考慮事項

COBOL で開発されたバッチアプリケーションの jBatch 化について唯一となる解やプラクティスはありません。予算や期間等の制約事項、移行時に実現したい改善事項、取り入れたい JVM 向け技術、利用するデータ等、様々な要素を総合的に判断して最適解を導出する必要があります。

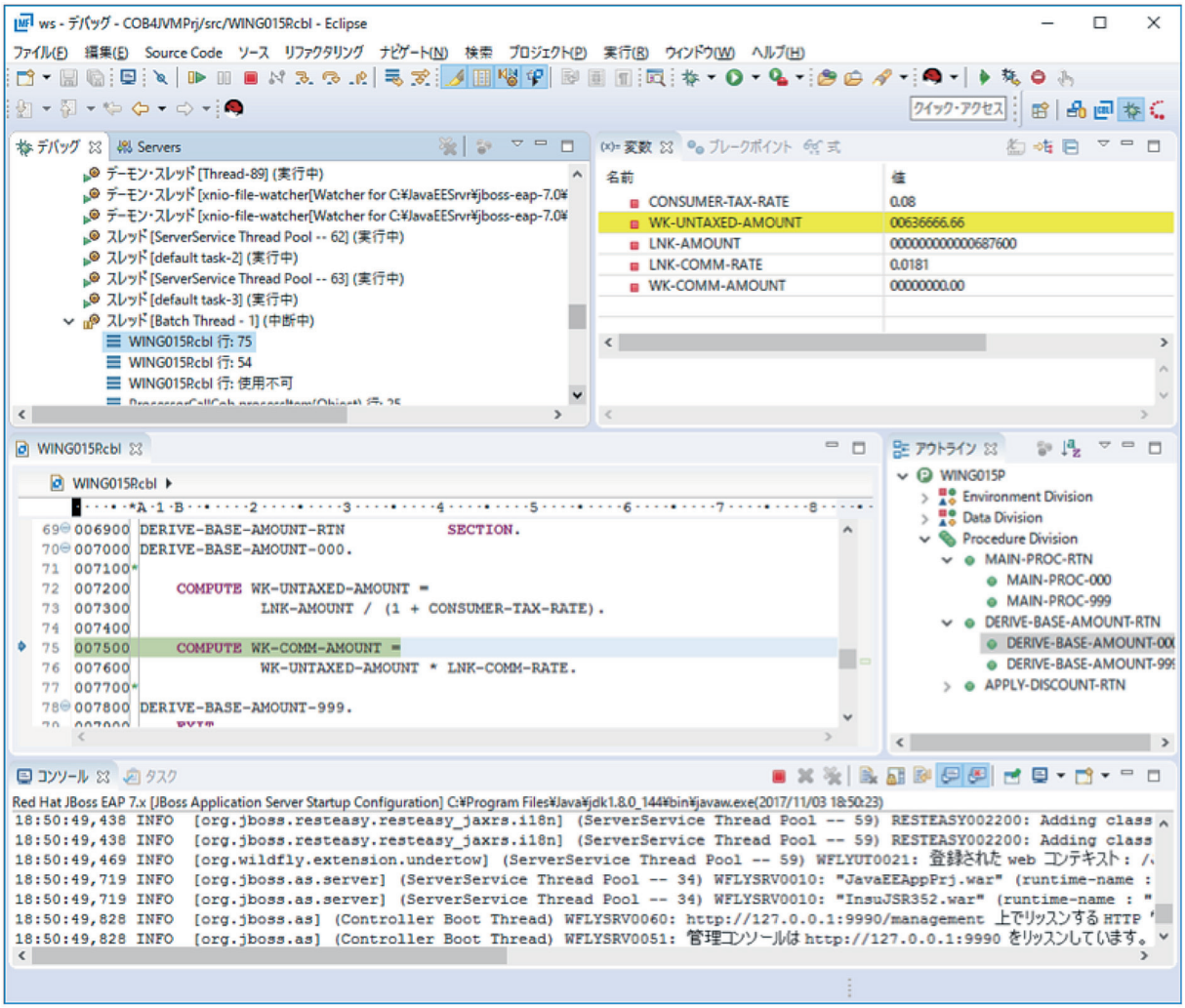
前章で紹介した内容以外にも jBatch には多くの機能が用意されています。これらを全て活用してリエンジニアリングすれば非常に高機能なアプリケーションへとモダナイズできますが、それなりにコストもかかる上、手を入れる分、品質劣化のリスクも伴います。逆に例えば第1段階の移行では、運用コストの削減や JVM 標準の運用技術の採用を最優先課題として、COBOL アプリケーションは原則手を入れずに Batchlet ステップのみで構成されるジョブへ移行するのも選択肢の1つです。

ただ、モダナイゼーションに当たってはコスト削減や運用改善のみならず、第 2 段階以降に業務改善も見据えているはずです。この業務改善事項も反映させてアプリケーションをリエンジニアリングするのであれば、より高機能な Chunk タイプのステップとして設計を見直すべきです。この際も全てを作り替えるのではなく、再利用できるものとできないもので仕分けが必要です。企業固有のビジネスルールを基に開発されたビジネスロジックについては業務改善後もアプリケーションのコア部分となるはずです。このビジネスロジックについて既に品質が担保されたプログラムがあり、それが COBOL で書かれているのであれば前項でみたように COBOL のまま Chunk ステップでも継続利用できますのでそれを Java 等で書き直すメリットはありません。前項の要領で Processor から COBOL で書かれたビジネスロジックを継続利用します。

この他の部分についてはプログラム言語の特徴を見極め、どの言語を利用するべきか適切に判断します。例えば十進演算や COBOL のデータファイル IO 等を伴う処理については作り直しになるにしても COBOL で書いた方が保守面、品質面からも効果的です。一方、Reader、Processor、Writer の実装ロジックについては COBOL でも書くことはできますが、書籍やネット上のサンプル等を参考にするのであれば Java で書いた方が効率的かもしれません。この場合、JVM アプリケーションであっても COBOL と Java が共存することとなり、管理の煩雑さが懸念されます。

しかし、Visual COBOL は Eclipse 上の開発をサポートしており、同一の Eclipse ワークスペース上で、COBOL と Java のプロジェクトを共存させることが可能です。更に、JVM COBOL プロジェクト中のソースは Java EE アプリケーションのデバッグ構成にてソース・ルックアップ・パスへ追加指定が可能です。つまり、Micro Focus が COBOL 向けに作りこんだデバッガーが Java EE アプリケーションのデバッグでも起動されるため、同じ Eclipse のワークスペース上で COBOL と Java の混合言語デバッグをシームレスに行えます。

図6- 8 Java EE アプリケーションデバッグにおける COBOL デバッグイメージ



jBatch は JVM 上で稼動する Java EE アプリケーション用のフレームワークです。策定にあたっては IBM 社のメンバーが指揮を執り仕様が練られた[15] こともあり、メインフレームにおけるバッチアプリケーション運用思想も反映されているようです。jBatch においてもジョブは複数のステップから構成され、その定義は JCL に相当するより高機能な JSL で行います。また、JBeret[16] をはじめとした jBatch の実装においてもメインフレームと同様にキューという概念が存在します。しかし、必ずしもメインフレームでできていたことが jBatch でも実装されているわけではありません。メインフレームから移行する場合はとりわけ下記の点には注意が必要です。

> ジョブクラスの概念

> 同名ジョブの連続実行抑止の仕組み

jBatch にはメインフレームにおけるジョブで指定するクラスという概念が存在しません。そのため、メインフレームのようにジョブクラス毎にイニシエーターを割り振ったり、キューにジョブが溜まった場合に優先順位の高いジョブから処理させていくといったことができません。ただ、jBatch の実装の多くはスレッドプールの数を構成可能にしています。つまりスケーラブルな環境で運用し、キューが溜まらないようスレッドプールをサーバーリソースと併せてチューニングすることで、本制約自体を不問にすることもできます。

あるジョブが実行中に追加で同じジョブが実行されますと、出力ファイルの排他機能が正しく構成されていなければ結果に矛盾が生じる可能性があります。この点についてはリスナー機能を活用することで回避が可能です。javax.batch.api.listener.JobListener インターフェイスを実装し beforeJob() メソッドを用意すると、ジョブの実行前にこのメソッドにコントロールが渡ります。このメソッド内で起動しようとするジョブと同じジョブが実行中であれば起動をホールドするような処理を入れれば、同じジョブが同時実行されることはありません。以上のようにメインフレームから移行する場合、必ずしも対応する機能が存在しないこともあります。しかし、それが JVM アプリケーションとして運用する上で本当に必要か、仮に必要であっても jBatch や JVM 向けの機能で代替できないか入念に分析した上で移行 feasibility 判断することが求められます。

15 参考「JSR 352: Batch Applications for the Java Platform」
参考キーワード
「Specification Leads: Chris Vignola, IBM、Maintenance Lead: Scott Kurz, IBM」
https://jcp.org/en/jsr/detail?id=352 (リンク確認 : 2018/02/14)

16 WildFly による jBatch の実装。Red Hat JBoss Enterprise Application Platform 7.0 よりバンドル。
参考「JBeret User Guide」
https://jberet.gitbooks.io/jberet-user-guide/content/ (リンク確認 : 2018/02/14)

7 パフォーマンス比較

7-1 検証内容

定められたタイムスロット内で処理を完了させることが必須であるバッチアプリケーションにとってパフォーマンスは重要な要素です。ここまで JCL を含めたリホスト、COBOL のみを対象としたリホスト、jBatch を活用した JVM アプリケーション化の手法や効果について解説してきましたが、それぞれの手法を採用したところでパフォーマンスを改善できるのか気になるところではあります。パフォーマンスについては、ハード資源やアプリケーションの作りに大きく依存するため、定性的に各手法の優劣を論じることは困難です。比較するのであれば、実アプリケーションを使って本番を想定した環境で検証を行い定量的な比較を行うべきです。

本章では本稿執筆にあたり Micro Focus 社が用意したメインフレーム上で稼動するデモアプリケーションをベースに各移行パターンに適用した上でパフォーマンス比較検証を実施しました。上述の通り、実際に移行する際には別途実アプリケーションにて検証し、非機能要件が満たせているか確認する必要があります。本章の結果に関しては、1つの参考ケースとしてご利用ください。

今回用意したメインフレーム上で稼動するアプリケーションは架空の損害保険会社が利用する年次の売り上げ集計アプリケーションとなります。アプリケーションは下表に示す2つのステップで構成されます。

表7- 1 検証で利用したアプリケーションの概要

Step 1		
	処理概要	・入力ファイルレコードにおける支店 ID を第 1 キーに、自動車保険成約月を第 2 キーとしてソートを行う ・ソートは COBOL で書かれたプログラム中の SORT 文で処理
	入出力データ	
	入力ファイル	・各支店で管理する自動車保険の成約情報を別のジョブでマージした順編成ファイル ・レコード中には保険の成約金額は含まれない
	出力ファイル	・本ステップの処理によりソートされたソート済みのファイル ・順編成ファイル
Step 2		
	処理概要	・入力ファイルを全件読み込み ・各レコード中の成約情報をもとに集計時点での保険料を算出 --> 保険料算出ロジックが埋め込まれたビジネスロジックプログラムを呼び出し ・支店毎に各月の成約金額を集計 ・会社全体の対象年月の成約金額を集計 ・集計情報を帳票へ出力
	入出力データ	
	入力ファイル	Step 1 の出力ファイル
	出力ファイル	支店毎の各月の成約金額及び全社の対象年月の成約金額が印字された帳票

本検証ではこのアプリケーションを下表の5つのパターンで動作できるよう環境とアプリケーションの調整を行った上で計測を行いました。パターン1はメインフレーム上で実施し、移行対象のレガシーアプリケーションを想定します。パターン2~5は本書で紹介した各移行手法を用いてモダナイゼーションを行ったパターンを想定しています。

表7- 2 パフォーマンス検証パターン

パターン1 ：レガシーアプリケーション	
実行環境	メインフレーム
ジョブコントロール	JCL
Step 1	COBOL
Step 2	COBOL
パターン2 ：JCL を含めたリホスト	
実行環境	クラウド環境 - Windows Server 2016
ジョブコントロール	JCL
Step 1	COBOL
Step 2	COBOL
パターン3 ：COBOL のみを対象としたリホスト	
実行環境	クラウド環境 - Windows Server 2016
ジョブコントロール	Windows PowerShell
Step 1	COBOL
Step 2	COBOL
パターン4 ：jBatch 化によるモダナイゼーション(Batchlet)	
実行環境	クラウド環境 - Windows Server 2016
ジョブコントロール	jBatch - JSL
Step 1	Batchlet ステップ中でパターン 1 ～ 3 で利用した COBOLプログラムを再利用
Step 2	Batchlet ステップ中でパターン 1 ～ 3 で利用した COBOLプログラムを再利用
パターン5 ：jBatch 化によるモダナイゼーション(Chunk)	
実行環境	クラウド環境 - Windows Server 2016
ジョブコントロール	jBatch - JSL
Step 1	Batchlet ステップ中でパターン 1 ～ 3 で利用した COBOLプログラムを再利用
Step 2	Chunk ステップで運用できるよう作り替え。ただし、パターン 1 ～ 3 で利用した保険料を算出する COBOL のビジネスロジック用のプログラムはそのまま再利用。

実行環境として利用したサーバーの詳細は下記となります。

表7-3 パフォーマンス検証パターン

＞ メインフレーム環境

ハードウェア	ソフトウェア
IBM z13 2965-N10 capacity model E03を利用 CPU 2965(3 CPUs) メモリー 32 M bytes OS Z/OS 02.02.00 MIPS 約 353	＞ IBM COBOL FOR OS/390 & VM 2.1.1 ＞ Z/OS DFSORT V2R2 ＞ JES2 Z/OS 2.2

＞ クラウド環境

ハードウェア	ソフトウェア
Microsoft Azure Cloud Computing Platform & Services を利用し東アジアリージョンにて Virtual Machine を展開 CPU　Intel Xeon CPU E5-2673 2.40 GHz 4 cores メモリー　7.0 G bytes OS　Windows Server 2016	＞ Micro Focus Enterprise Developer 3.0J for Windows ＞ Windows PowerShell 5.1（パターン3でのみ利用） ＞ Red Hat JBoss Enterprise Application Platform 7（パターン4、5でのみ利用） ＞ Java SE Development Kit 8u144（パターン4、5でのみ利用）

7-2 検証結果

本検証ではアプリケーションのステップ1で利用する入力データとして 10 MB、20 MB、200 MB とサイズが異なるファイルを3つ用意しました。これを各検証パターンでそれぞれ5回ずつ実行し、その平均値をもとに比較を行っています。その計測結果をまとめたものが下表となります。

表7-4 パフォーマンス検証結果

	10 MB	20 MB	200 MB
パターン1	0:00:06.800	0:00:15.800	0:02:24.600
パターン2	0:00:01.600	0:00:02.600	0:00:21.800
(対 パターン1)	23.53%	16.46%	15.08%
パターン3	0:00:01.232	0:00:02.429	0:00:21.310
(対 パターン1)	18.11%	15.37%	14.74%
パターン4	0:00:01.933	0:00:03.981	0:00:35.988
(対 パターン1)	28.42%	25.20%	24.89%
パターン5	0:00:02.075	0:00:04.049	0:00:40.041
(対 パターン1)	30.51%	25.63%	27.69%

今回紹介した移行パターン（パターン２～4）に関してはいずれもパターン1のレガシーアプリケーションと比べて高速に処理できています。いずれのパターンについても処理時間が入力データサイズとほぼ比例していることからこの関係は入力データのサイズ関わらず維持されることが推測されます。CPU ネイティブの機械語にコンパイルされるパターン2及び3に関してはレガシーからのパフォーマンス改善が特に顕著です。レガシーアプリケーションと比べて約6倍のスピードで処理できているようです。パターン3に関してはパターン2の JES のエミュレーションによるオーバーヘッドがない分、より高速です。パターン4及び5は Java bytecode という中間コードで構成されるモジュールを利用します。中間コードとなるため、パターン2及び3のような事前コンパイル形式と異なり、実行時に JVM 上の Just In-Time Compiler が CPU ネイティブの機械語をメモリー上に生成して実行します。この分のオーバーヘッドがあるため、COBOL を含む含まないに関わらず JVM アプリケーションに関しては事前コンパイルによるアプリケーションと比べるとどうしてもパフォーマンスの点では不利になります。しかし、今回の検証結果を見ますと、それでもレガシーと比べて約4倍のスピードで処理できていることが確認できます。Chunk タイプのステップを含めたパターン5についてはチェックポイントによるオーバーヘッドがある分、パターン4と比べて若干のパフォーマンス劣化が確認できます。

おわりに

本書では、ビジネスのデジタルトランスフォーメーションが不可欠な昨今において、テクノロジー革新が先送りにされる傾向のあるバッチアプリケーションのモダナイゼーション手法を３つ紹介しました。いずれの手法に関しても確かな稼動実績のある COBOL モジュールを再利用しつつも刷新に取り組むことが可能になります。

第３章、第４章で紹介した「JCL を含めたリホスト」や「COBOL のみを対象としたリホスト」はプラットフォームや利用ツールを刷新することで TCO の改善、パフォーマンスの改善、開発・運用メソドロジーの改善を図ろうというものです。TCO の改善はデジタルビジネスのような攻めの IT 投資力強化につながります。これらの手法ではアプリケーションはネイティブコードとしてコンパイルされているため、パフォーマンス改善を主目的とするのであれば最も効果が期待できます。スケーラブルな環境での運用を可能とするため、メインフレームからの移行のみならず、既にオープン化されているケースでもプラットフォーム刷新による TCO 及びパフォーマンスの改善は同様に期待ができます。更に最新の開発・運用製品に切り替えることでアジャイル開発や DevOps に取り組むことも可能です。

第５章、第６章で紹介した「jBatch 化によるアプリケーションのモダナイゼーション」についてはリホストソリューションによる改善事項はもちろんのこと、更にアプリケーションを JVM 環境へ開放してアプリケーションそのものの刷新をも可能にするソリューションとなります。JVM アプリケーションとして運用することで JVM 向けの技術との融合が可能になるほか、Java EE アプリケーションとして運用することのメリットも享受できるようになります。既に Java で書いた自社アプリケーションがあるのであれば、それらとの連携も図ることができます。JVM アプリケーションに関しては中間コードとなるため、ネイティブのアプリケーションと比べてパフォーマンスに関する懸念はつきまといます。しかし、条件によっては第7章で紹介したパフォーマンス検証結果のように移行前環境と比べて大幅なパフォーマンス改善効果が得られる場合もあります。

以上のようにバッチアプリケーションのモダナイゼーションに関しても既に様々な選択肢が揃っています。評価項目、優先順位、制約事項等によってその時点での最善の選択肢は異なります。しかし、アプリケーションが常に進化し続ける必要があることを考えると先を見据えた判断が肝要となります。Micro Focus はこの先を見据えた使い方に重点を置いて製品設計しています。リホストソリューションを選択されても、同じ製品で JVM によるアプリケーションモダナイゼーションへ取り組むことを可能にしています。JVM 化した後も Batchlet から Chunk 化への進化を支援する JVM 向け拡張方言も提供しています。移行を検討される際はこのモダナイゼーションパスにも是非注目してください。本書がバッチアプリケーション刷新を目指される皆様にとって検討の一助となれば幸いです。

マイクロフォーカス合同会社
www.microfocus.co.jp