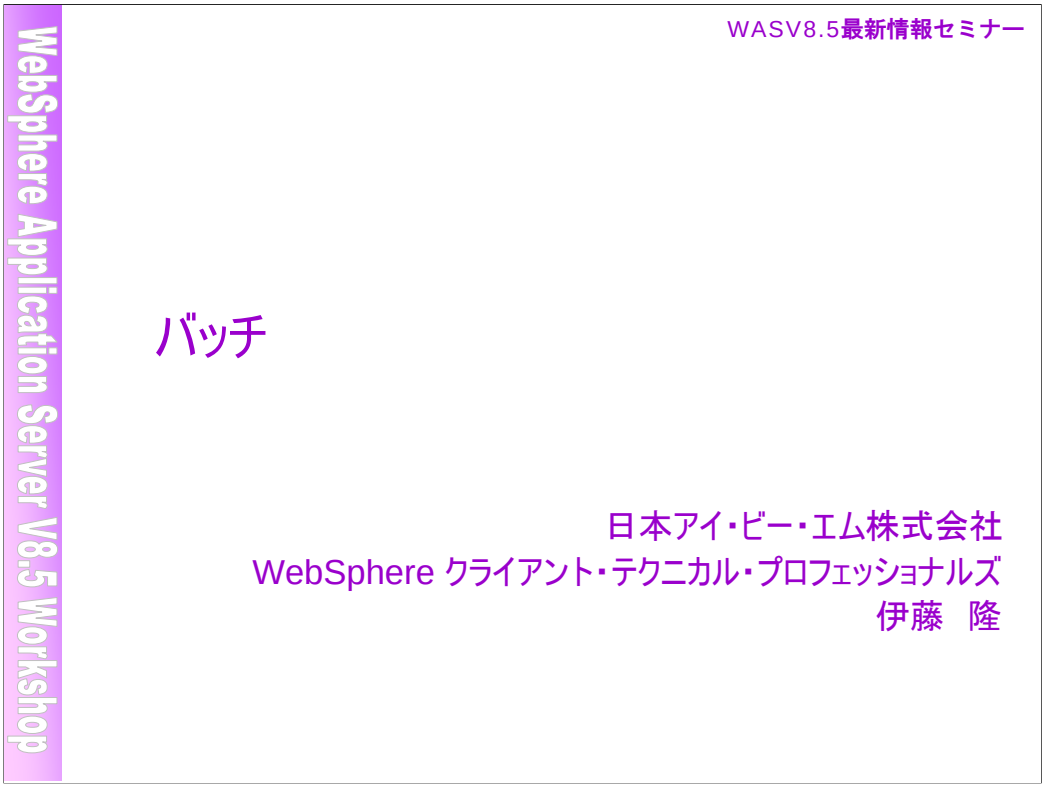


当セッションでは、WASv8.5で大きく強化されたバッチの機能について、お話いたします。

WebSphere Application Server V8.5 Workshop

Agenda

1. WebSphereバッチ概要

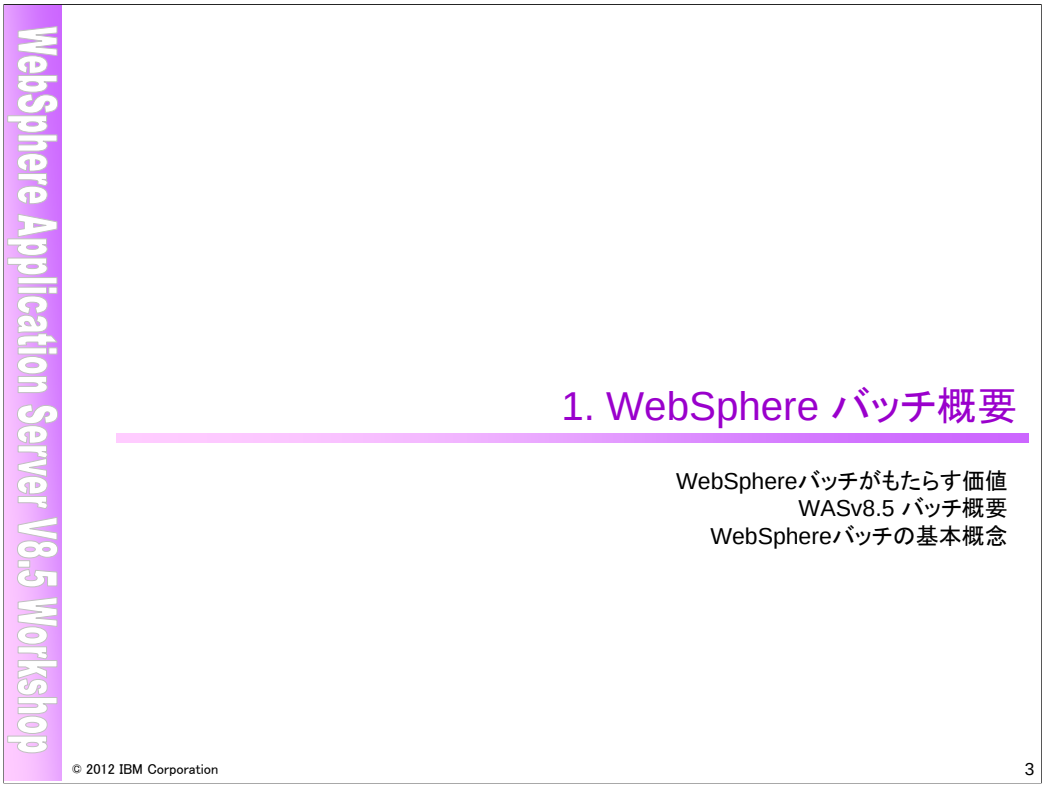
2. プログラミング・モデルと開発環境

3. ジョブの実行と管理

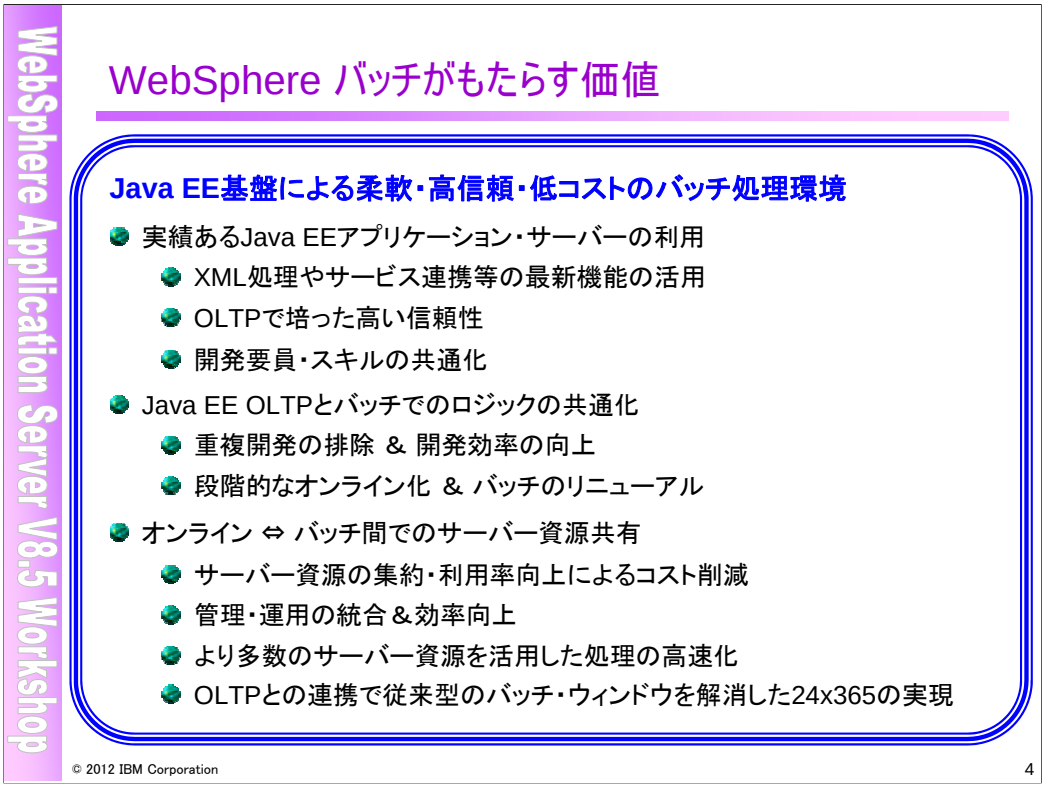
4. まとめ

参考文献

概要に続いて、内容は大きく2つのパートでお話します。
前半は、開発関連のトピック、後半が、実行環境に関連するトピックになります。



それでは、まず、WAS V8.5で統合＆強化されたバッチ機能の概要についてお話します。



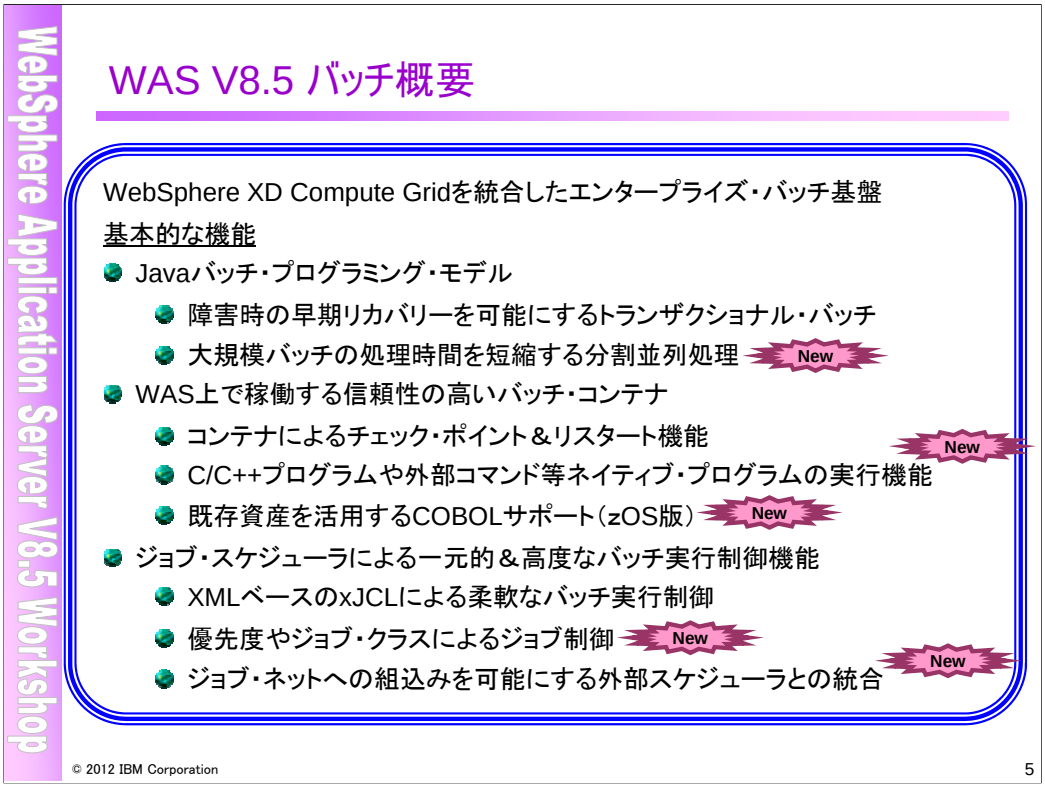
WASv8.5のバッチ機能は、Java EE基盤による高度で柔軟なバッチ環境を提供します。では、このようなJava EEによるバッチ基盤は、お客様に、どのような価値をもたらすでしょうか？ ここでは、大きく3つの観点から、期待される効果を挙げたいと思います。

1つ目は、現在のJava EEサーバー製品が、最新機能を備え、かつ、高い信頼性を持つソフトウェアであり、広く使用されていることで非常に大きなスキル・ベースを持つ点です。いまや、Java EE製品は、企業の基幹系業務でも利用されるほど、信頼性が高く、かつ、企業のあらゆるシーンで利用されており、これをオンラインだけでなく、バッチでも利用しようというのは、当然の流れとも言えます。

2つ目は、OLTPとのコードの共通化がもたらす利点です。重複開発を排除して効率を向上するといった側面に加えて、バッチをオンライン化していく上で、コードを共有しながら段階的な移行が可能になります。

3つ目は、サーバー資源が共有できるという点です。バッチ用か、オンライン用かを問わず、リソースを共有して柔軟に利用することができることで、コストや効率が向上できます。また、オンラインの空き資源を活用することで、バッチ処理を高速化し、バッチ・ウィンドウを短縮できるといった効果も期待できます。更に、OLTPとより密に連携することで、相当量のデータを溜め込んでバッチ専用システムで処理するのではなく、随時少量のデータを短時間で処理して、いわゆるバッチウィンドウの解消を図ることも可能になります。

このように、様々な利点があり、実際に多くのお客様で、実際にJava EE基盤をバッチ処理を活用しようと検討、或いは、実際に活用しています。



前ページで挙げたような価値に着目して、約7年前に、IBMは、WASのアドオン製品としバッチ製品をリリースしました。

その後、コアのバッチ機能を、V7のFeature Packに、次いで、WASv8に搭載してきましたが、遂に、今回リリースされたWASv8.5において、全バッチ機能が統合されました。このように、7年間もの間、進歩を重ねてきたバッチ機能は、先に上げた価値＆効果を更に高める諸機能を提供します。

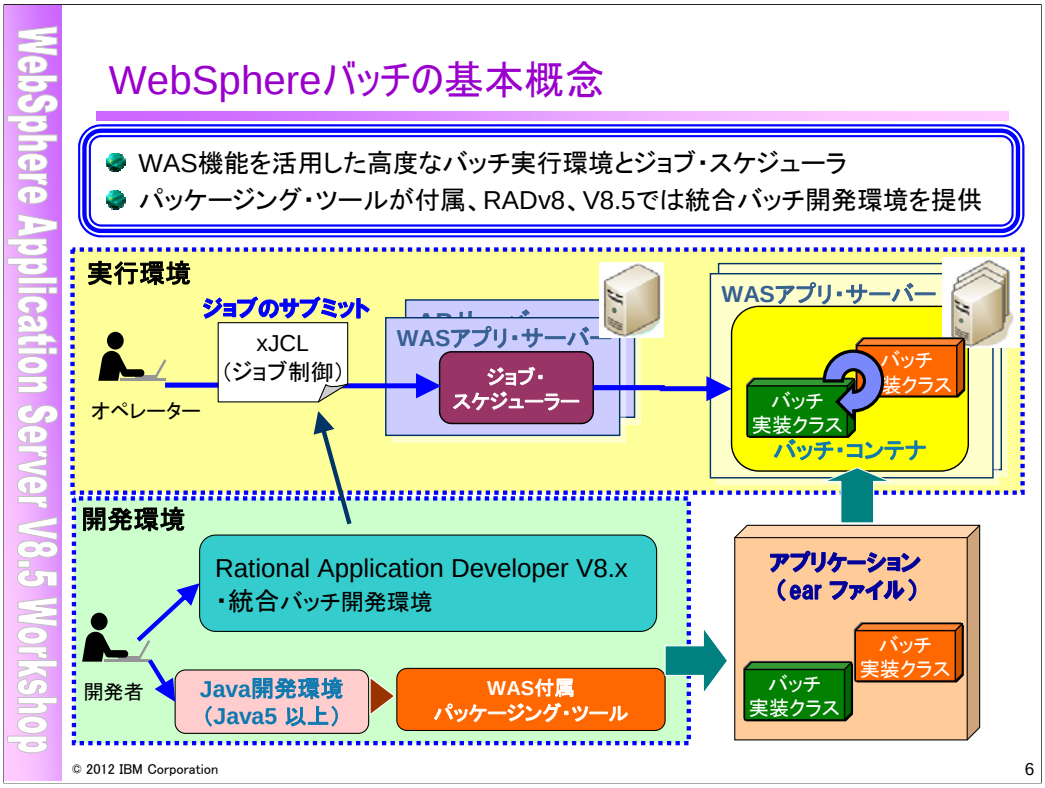
ここでは、その主な機能を簡単に紹介します。詳細は、2章、3章でご説明していきます。

まず、プログラミング・モデルに基づくJavaバッチの実現です。プログラミング・モデルを通じたランタイムとの連携により、チェックポイント・リスタートや、並列処理といった機能を実現します。

2番目は、ランタイム環境であるバッチ・コンテナの機能です。信頼性に加えて、Java以外の言語で開発されたコマンドやユーティリティといったネイティブ実行機能や、zOS版に限りますが、COBOLの実行環境も提供します。

3番目は、ジョブ・スケジュラーの機能です。ジョブが一元的に管理できるほか、優先度制御やジョブクラスといった高度な機能が利用できます。

また、企業で既に利用されている実績のあるTivoli Workload Schedulerのようなエンタープライズ・スケジューラから、ジョブを実行し連携するユーティリティ機能を提供します。これによって、企業内のジョブ・ネットに、WASによるバッチ基盤を組み込むことができます。



詳細の説明に入る前に、WASv8.5のバッチ機能を実現するためのコンポーネントや開発ツール等を俯瞰してみましょう。

まず、左下にあるのは開発環境です。

RADは、v8.0.4以降で、WebSphereバッチの開発機能を提供しています。これを利用すれば、ウィジェット形式で、バッチ開発を行い、パッケージング＆テストまで行うことができます。

また、WASv8.5には、パッケージング・ツールが付属しますので、使い慣れたJava開発環境で開発を行い、それをパッケージング・ツールでパッケージングすることも可能です。

開発したアプリケーションは、拡張されたearファイルとしてパッケージングします。

また、開発者は、ジョブの実行者に対して、xJCLを作成するための雛形を提供する必要もあります。

RADでは、このxJCLを作成するxJCLエディタが付属します。

実行に際しては、まず、アプリケーションのパッケージであるearファイルをWASにデプロイする必要があります。バッチ・アプリケーションを含むearファイルを、WASにデプロイすると、アプリケーション・サーバーはバッチ・コンテナを用意し、サーバー再起動後、利用が可能になります。

バッチ・コンテナ上で、アプリケーションが稼動できる状態になったら、オペレーターは、バッチの実行内容をxJCLに記述して、ジョブ・スケジューラにサブミットします。ジョブ・スケジューラは、アプリケーション・サーバーの中から適切はサーバーを選択して、ジョブをディスパッチします。

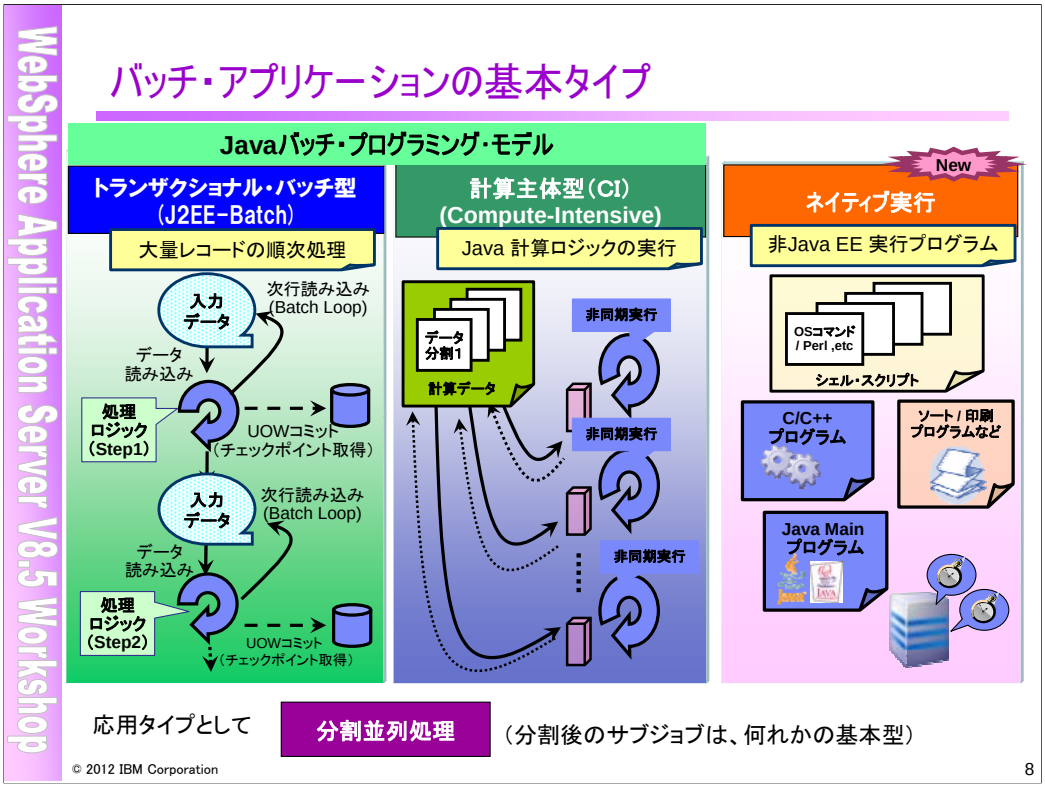
ディスパッチされたサーバーは、xJCLに基づいてバッチを実行します。

これが、開発を含めた、WASv8.5のバッチに関連する主要概念になります。

WebSphere Application Server V8.5 Workshop

2. プログラミング・モデルと開発環境

バッチ・アプリケーションの基本タイプ
トランザクショナル・バッチ型
ネイティブ実行
ジョブの分割＆並列処理



WASv8.5がサポートするバッチ・アプリケーションには、大きく分けて3つの基本形があります。

また、それらとは別に、分割並列処理がありますが、分割されて実行されるサブジョブは、これらJavaバッチ・プログラミングの基本形に属します。

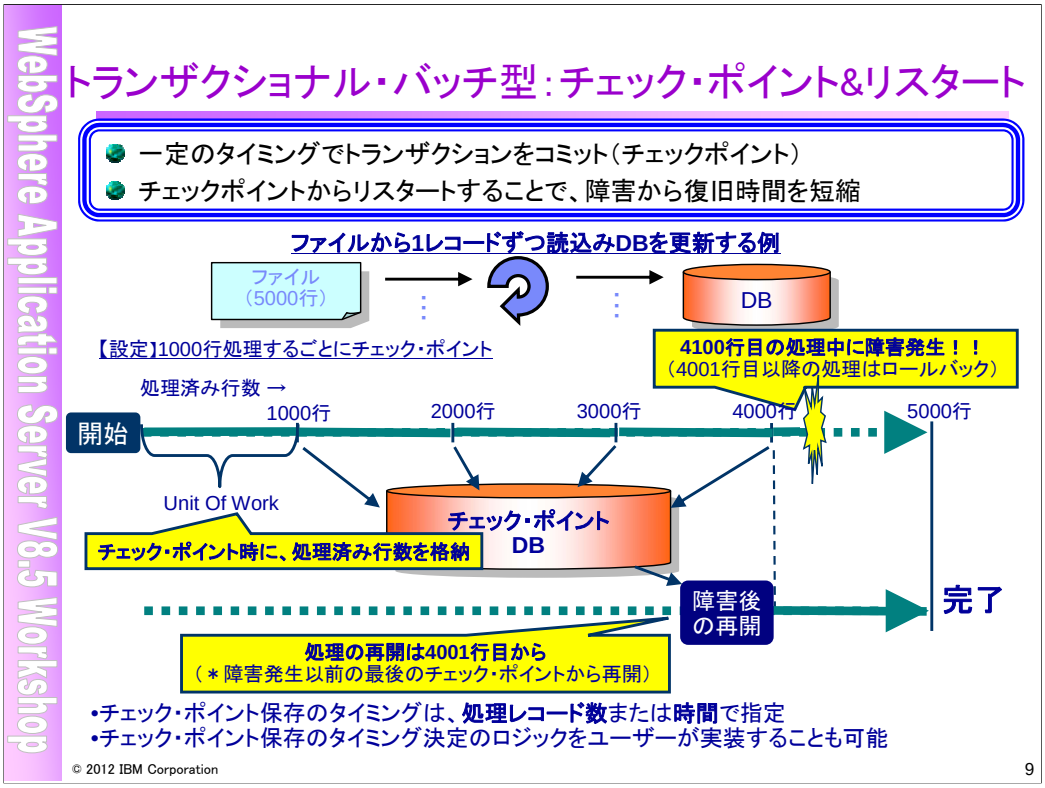
まず、トランザクショナル・バッチ型は、データを1レコードずつ入力して処理を行うタイプで、処理中に、一定のタイミングで、チェックポイントをDBに記録します。開発者は、基本的に1レコードの処理を記述し、繰り返しのループ制御などは記述しません。レコード処理の繰り返しはコンテナが行います。また、ここにあるように、ジョブには複数のステップが記述できるので、前のステップでの処理結果の出力データを、次の処理のインプットにして処理することが可能です。WASv8.5で、Javaバッチといった場合、狭義には、このトランザクショナル・バッチ型を指す場合があります。

次の計算主体型は、逆に、コンテナからは、run()というメソッドを呼び出すだけで、あとの制御は全てユーザー・プログラムに委ねられます。データの存在もコンテナからは見えません。したがって、データ主体ではなく、計算主体向けということで、この名称になっています。

3つ目のタイプは、ネイティブ実行型、つまり、Javaとも限らず、C/C++で作成された実行プログラムであったり、OSや他のSWに付属のコマンド・ツールや、或いは、単独で実行するJava Mainタイプのプログラムなどを実行するものです。

これによって、既存の資産と連携することで、開発のワークロードを圧縮しつつ、より高度なバッチ処理を実現できます。

以降、しばらくは、トランザクショナル・バッチについて詳細を見ていきます。



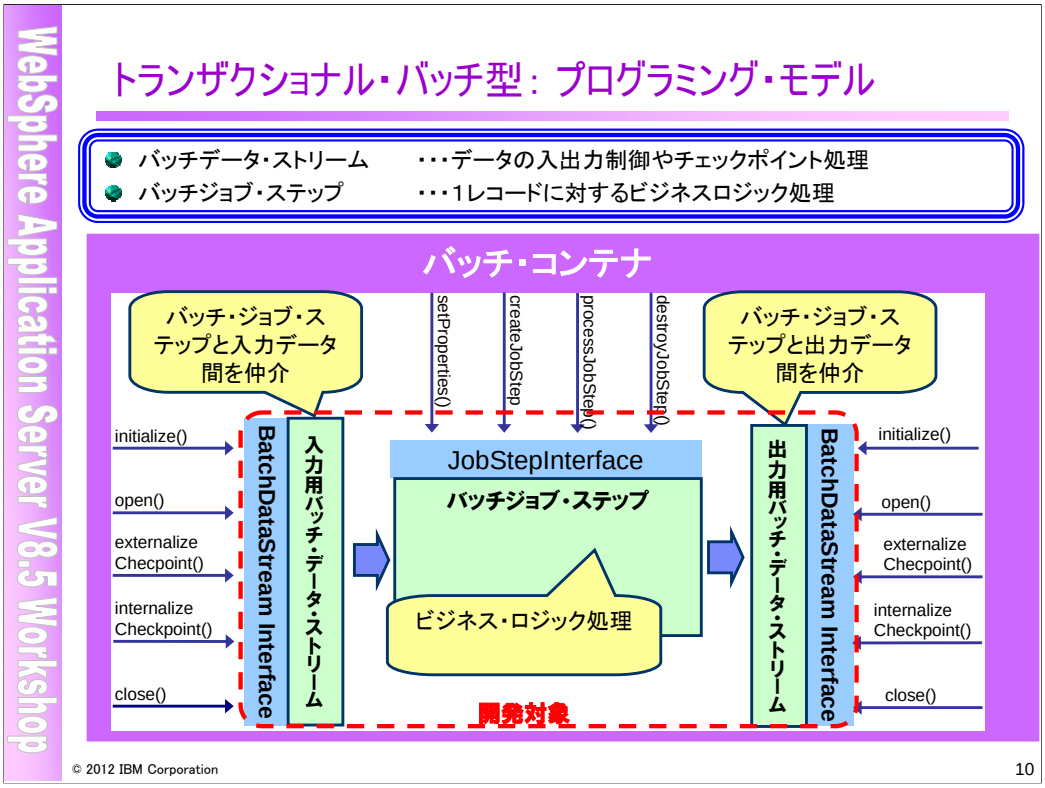
トランザクショナル・バッチについて、そのチェック・ポイント&リスタート機能について補足します。

コンテナは、バッチ処理において一定のタイミングで、トランザクションをコミットして、DBにチェックポイントを書き込みます。したがって、コミット時に他のDBへの更新が発生する処理では、グローバル・トランザクションになります。ただし、チェックポイントと同じDBへの更新であれば、ローカル・トランザクションを選択することもできます。

障害が発生すると、チェックポイントから、処理を再開します。これによって、それまでの処理を無駄にせず、処理再開から完了までの時間を大幅に短縮できます。

例えば、この図のように、1000行ごとにチェックポイントをとる設定とした場合、4100行目で障害が起これば、処理は4001行目から再開され、4000行までの処理は省くことができます。

チェックポイントのタイミングが重要になってきますが、これは、xJCL中に、レコード数、或いは、時間のどちらかで指定することができます。



次に、トランザクショナル・バッチを、どう開発・実装するかを、もう少し詳しく見てみましょう。

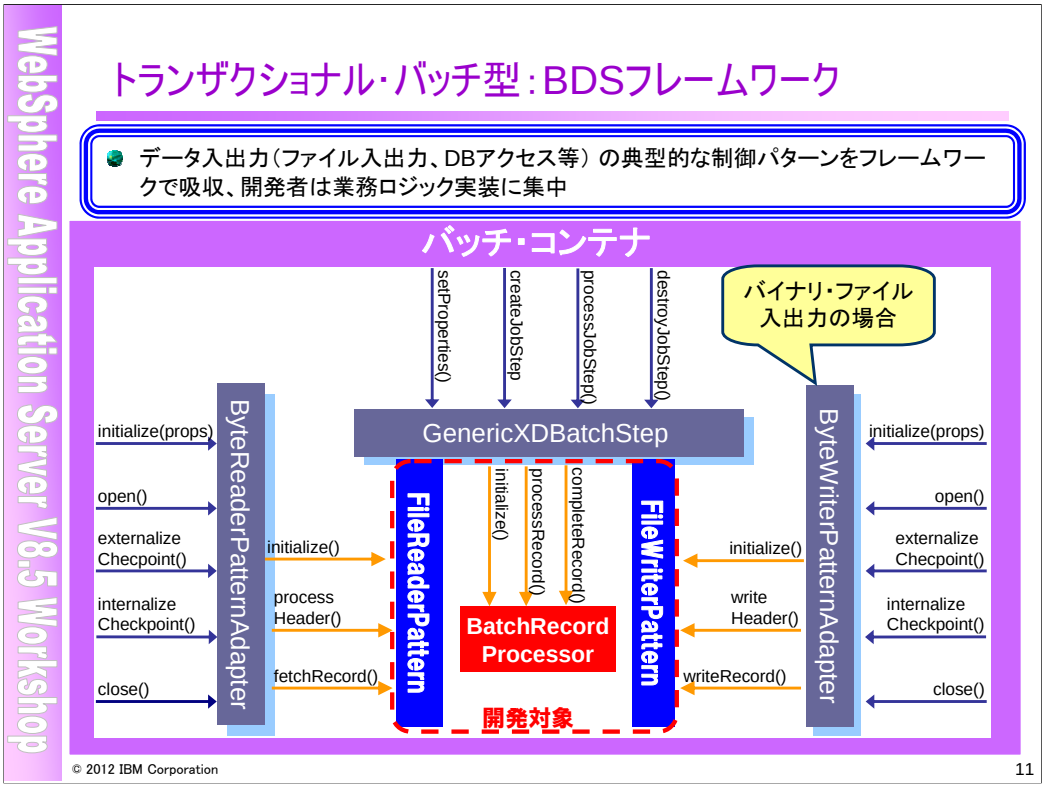
トランザクショナル・バッチ型のプログラミング・モデルは、大きく2つ、処理内容を記述したバッチ・ジョブ・ステップと、データの入出力を担うバッチ・データ・ストリームの2つのJavaコンポーネントからなります。

この図は、1レコード入力に対して処理を行い、1レコード出力すると言う典型的なケースでの、開発対象コンポーネントを表しています。この場合、1つのバッチ・ジョブ・ステップに対して、入力用、出力用のバッチ・データ・ストリーム、合計3つのコンポーネントを開発することになります。バッチ・データストリームや、バッチ・ジョブ・ステップは、ここにあるような、バッチ・コンテナとのインターフェースとなる各メソッドを実装する必要があります。

たとえば、JobStepInterfaceのprocessJobStep()は、まさに、レコード処理のためのメソッドで、終了のフラグが返るまで、繰り返し呼ばれます。

また、一定のタイミングで、バッチデータストリームに対して、externalizeCheckPoint()が呼ばれるので、チェックポイント情報、例えば、処理行数等、をコンテナに渡す必要があります。

インターフェースさえ実装すれば、ジョブ・ステップと、バッチ・データ・ストリーム間にやり取りなどの開発は、かなり自由度が大きいモデルとなっています。ただし、バッチコンテナとのインターフェースの実装が、かなり大変です。



BDSフレームワークは、幾つかの定型パターンについて、実装を提供し、開発の手間を大幅に削減します。

たとえば、入出力がそれぞれひとつのバイナリファイルであり、それぞれから、1レコード入力して1レコードを出力する処理であれば、次のようなパターンが適用できます。

それぞれ、インターフェース実装は、バイナリファイルの入力については、ByteReaderPatternAdapterを、出力には、ByteWritePatternAdapter、そして、バッチ・ステップについては、GenericXDBatchStepが提供されます。そして、開発は、それぞれから呼ばれるFileReaderPattern、FileWritePattern、BatchRecordProcessorを実装することになります。単純に見ても、14個のメソッドが、9個のメソッドで済むというだけでなく、例えば、このファイルのリード＆ライトのケースでは、ファイルのオープンや読込み、クローズといった定型的な処理は、パターン実装に含まれ、開発者が実装する必要がありません。

このように、BDSフレームワークによって、開発のワークロードを大幅に軽減することが可能です。

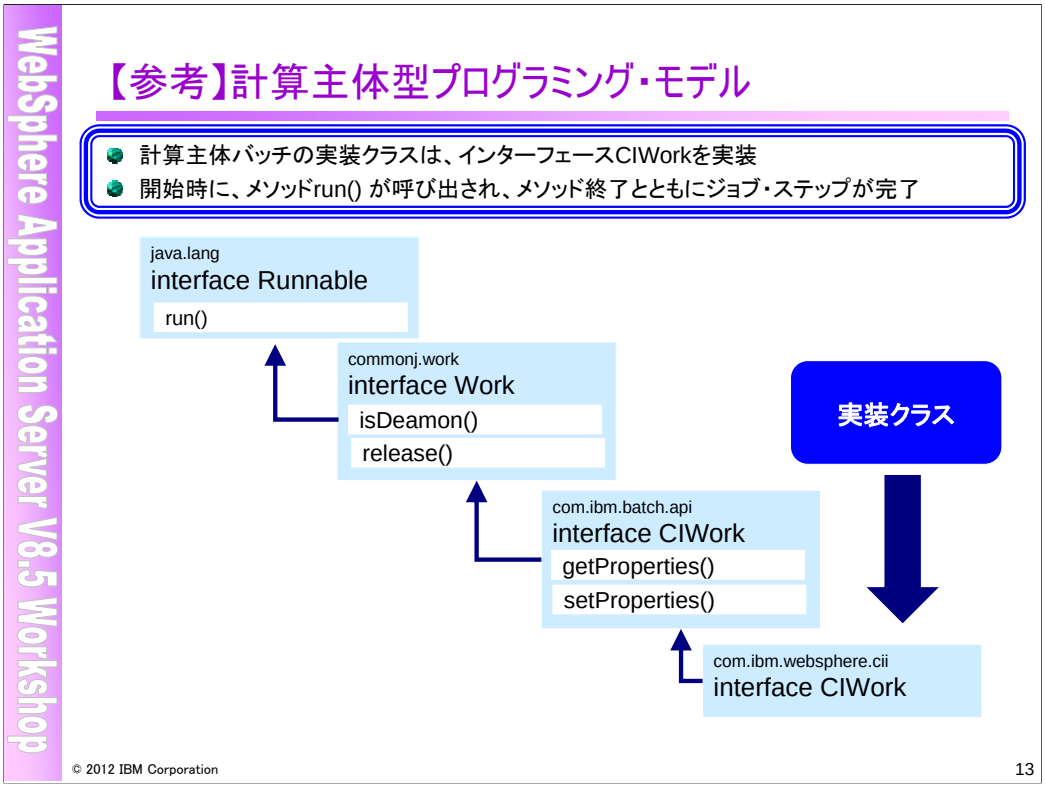
WebSphere Application Server V8.5 Workshop

【参考】BDSフレームワーク：提供されるパターンとクラス

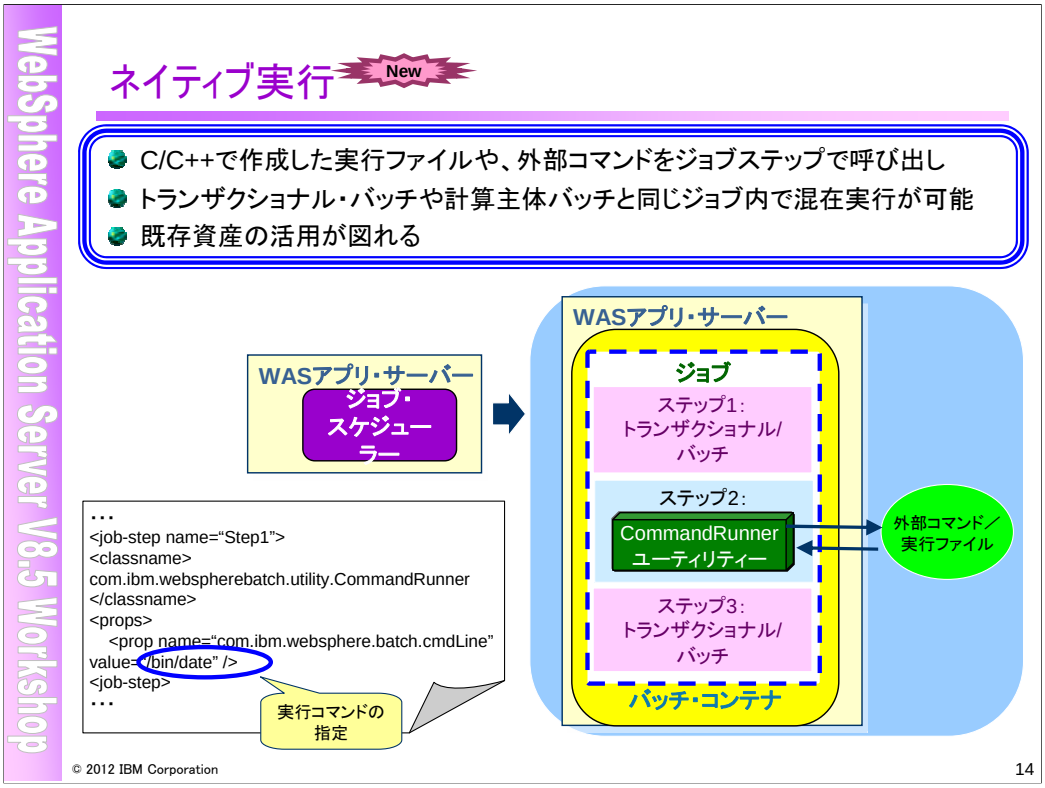
パターン名	説明	サポート・クラス
JDBCReaderPattern	JDBC 接続を使用してデータベースからデータを取得する場合に使用されます。	LocalJDBCReader JDBCReader CursorHoldableJDBCReader
JDBCWriterPattern	JDBC 接続を使用してデータベースにデータを書き込む場合に使用されます。	LocalJDBCWriter JDBCWriter
ByteReaderPattern	ファイルからバイト・データを読み取る場合に使用されます。	FileByteReader
ByteWriterPattern	ファイルにバイト・データを書き込む場合に使用されます。	FileByteWriter
FileReaderPattern	テキスト・ファイルの読み取りに使用されます。	TextFileReader
FileWriterPattern	テキスト・ファイルへの書き込みに使用されます	TextFileWriter
RecordOrientedDataSet-ReaderPattern	z/OS データ・セットの読み取りに使用されます	ZFileStreamOrientedTextReader ZFileStreamOrientedByteReader ZFileRecordOrientedDataReader
RecordOrientedDataset-WriterPattern	z/OS データ・セットへの書き込みに使用されます。	ZFileStreamOrientedTextWriter ZFileStreamOrientedByteWriter ZFileRecordOrientedDataReader
JPAReaderPattern	OpenJPA を使用してデータベースからデータを取得する場合に使用されます。	JPAReader
JPAWriterPattern	Java Persistence API (JPA) 接続を使用してデータベースにデータを書き込む場合に使用されます	JPAWriter

＊InfoCenterより転載

これらは、InfoCenterの抜粋で、BDSフレームワークが提供している入出力のパターンになります。
ファイル、DBのほか、zOSのデータセット用のパターンなども用意されています。



これも、参考までに、計算主体型の場合に実装するインターフェースです。3階層になっていますが、Java EE非同期の仕組みcommonj.workを拡張したシンプルなプログラミング・モデルで、開発者は、実質的にrun()メソッドのみを実装すれば良いことになります。



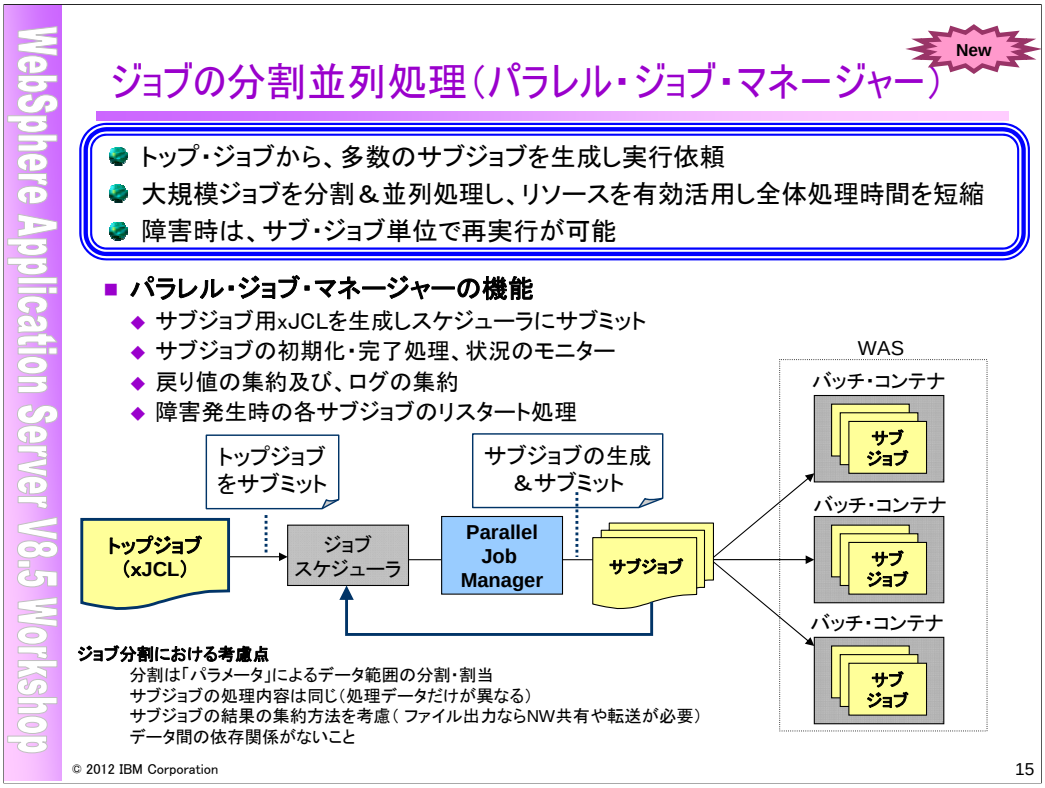
残る、もうひとつの基本型が、ネイティブ実行型です。

ジョブ中のジョブ・ステップの一環として、ネイティブのコマンドを実行するCommandRunnerユーティリティを使用できます。つまり、1ジョブの中で、トランザクショナル・バッチや、計算主体型バッチと連携する形で、ネイティブの実行を行うことができます。

例えば、最初のジョブ・ステップが、トランザクショナル・バッチで、DBのテーブルの各行を処理してファイルを出力し、

次に、CommandRunnderを使って、LinuxのSORTコマンドを呼び出して、ソートを行って、後続のトランザクショナル・バッチに渡すといった処理を1つのジョブとして行うことができます。

ちなみに、ネイティブ・ジョブの実行は簡単です。基本的に実行ファイルをxJCLに記述するだけです。オプションで、引数を与えたり環境変数を設定することもできます。



開発編の最後に、パラレル・ジョブ・マネージャーによるジョブの分割並列処理について簡単にご紹介します。

大規模なバッチ処理を行う場合、分割・並列処理による高速化が、当然必要となってきます。これを自動化する機能が、Parallel Job Manager（PJM）です。PJMは、トップジョブがサブミットされると、xJCLに記載されたパラメータを分割するためのクラスを呼び出し、指定された分割数のサブジョブ用xJCLを生成して、サブミットします。さらに、PJMは、これらのジョブ実行をモニターし、完了すると、戻り値、及び、ログを集約して、ひとつのジョブとして完了します。サブ・ジョブが失敗した場合でも、トップジョブを再始動することで、該当するサブジョブだけを再サブミットします。

このように、PJMは、一元的にジョブの分割・実行・モニターの機能を提供します。

WebSphere Application Server V8.5 Workshop

【参考】プログラミング・モデルの拡張

New

- OSGiバッチ・アプリケーション
  - バッチをOSGiアプリケーションとしてパッケージ化してデプロイ可能
- ジョブ・リスナー
  - ジョブ及びステップに関して初期設定とクリーンアップを追加する com.ibm.websphere.batch.listener.JobListener インターフェースの実装を提供
- ジョブ・ステップ・コンテキスト
  - 現行バッチ・ジョブ・ステップが実行されるコンテキストを一意的に識別する情報（ジョブID等）へのアクセス
  - アプリケーション固有情報を、バッチ・プログラミング・フレームワーク・メソッド間で受け渡せるユーザー・データ域
  - ステップをまたがってアプリケーション固有情報を渡すことができる一時的なユーザー・データ域
  - チェックポイント／再始動をまたがって、アプリケーション固有情報を保管する永続的なユーザー・データ域
- トランザクショナル・バッチの拡張
  - レベル・スキップ処理：レコードの読み取り、及び、書き込みエラーをスキップ
  - ステップ再試行処理：processJobStepメソッドにエラーが発生した場合に、ジョブ・ステップを再試行
  - トランザクション・モード：グローバル・トランザクション・モードかローカル・トランザクション・モードかを指定
  - バッチ・データ・ストリームのタイムアウト：トランザクション・タイムアウト（秒）を指定可能

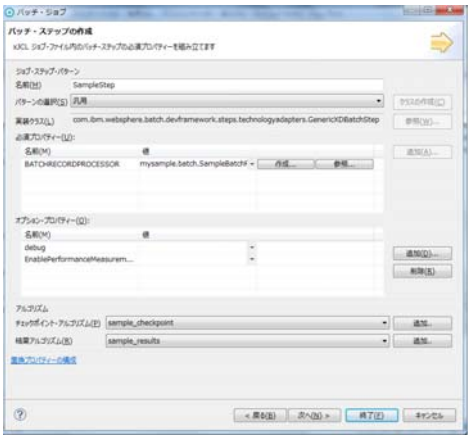
参考までに、ここまでに触れなかったプログラミング・モデルや開発関連の強化項目をリストしています。時間の都合で、説明は割愛させていただきます。

WebSphere Application Server V8.5 Workshop

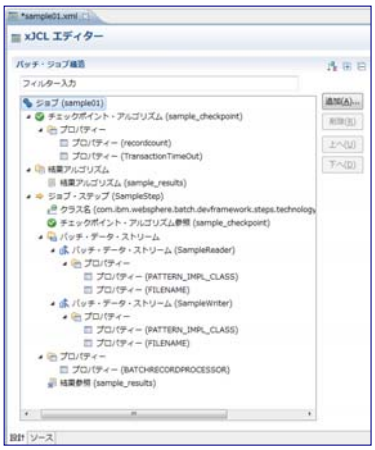
【参考】Rational Application Developer Javaバッチ開発

- ジョブ構成と必要なコンポーネントの作成を支援するジョブ作成ウィザード
- グラフィカルにxJCLを編集できるxJCLエディター
- 作成したバッチを、その場で稼働確認可能

ジョブ作成ウィザード：バッチ・ステップの作成画面



xJCLエディター

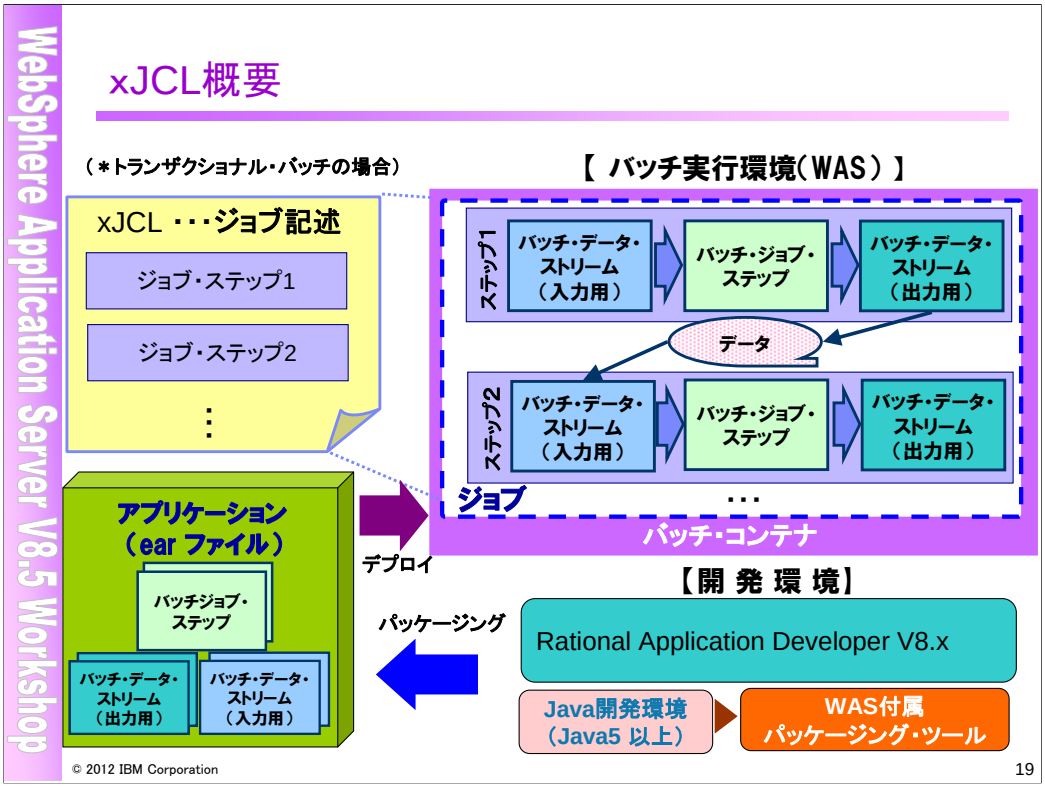


開発ツールの説明も割愛します。参考までに、Rational Application Developer for WebSphereの画面イメージを添付します。

WebSphere Application Server V8.5 Workshop

3. ジョブの実行と管理

xJCL概要
バッチ実行環境の概要
ジョブ管理コンソール
日時指定によるジョブ実行
ジョブ・クラス
ジョブ優先度制御
外部スケジューラ連携
ロギング機能



バッチ基盤のランタイムの説明の前に、バッチ実行の制御のための言語であるxJCLについてご説明します。

開発したバッチ実行のためのコンポーネント、Javaクラスは、 Rational Application Developer、或いは、WAS付属のツールで、earファイルとしてパッケージングし、WASで稼動するバッチコンテナにデプロイします。

デプロイした時点では、どのようなバッチ処理を、行うかはわかりません。処理の内容はxJCLで記述し、バッチ・コンテナに指示します。

xJCL では、どのコンポーネントをどう組み合わせて、どういった順番で実行するかを記述します。トランザクショナル・バッチであれば、幾つかのバッチ・データストリームと、バッチ・ジョブ・ステップを組み合わせて、一つのジョブ・ステップを構成します。このようなジョブ・ステップが1つ以上集まって、一つのジョブとなります。

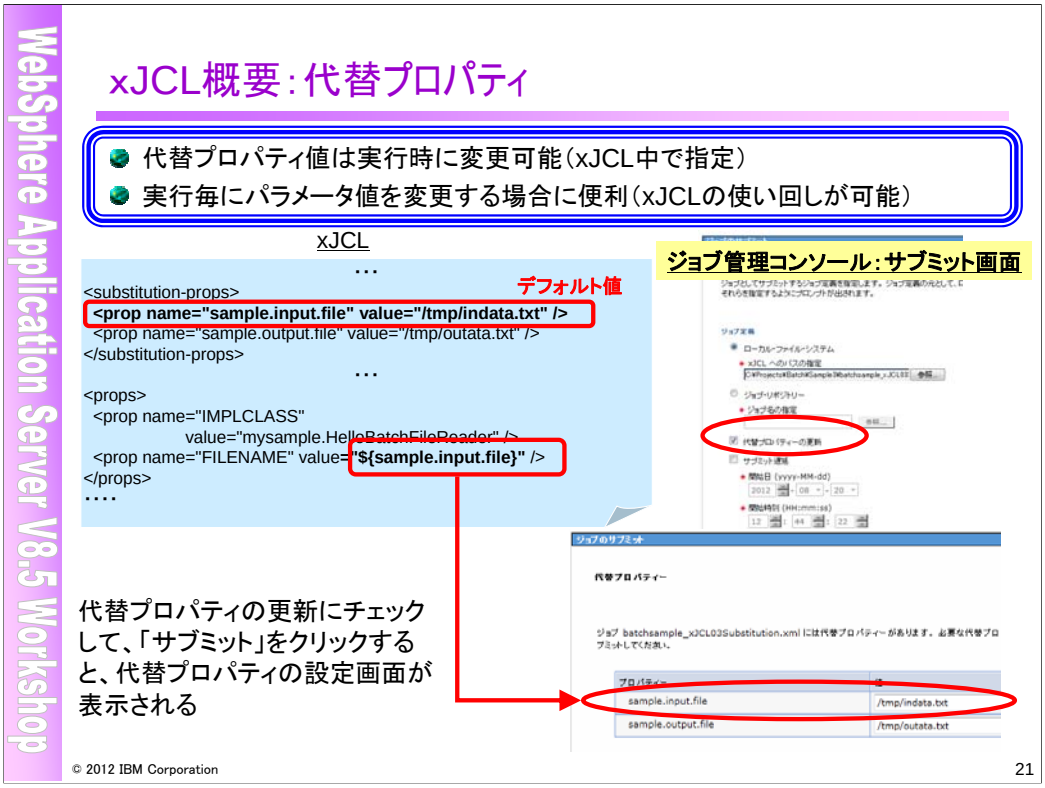
例えば、最初のジョブ・ステップでは、DBからデータを取り出して処理を行い結果をファイルに出力し、次のジョブ・ステップでは、このファイルを入力として、1行ずつ処理を行い、結果を、更に別のファイルに出力します。更に3番目のジョブ・ステップが、このファイルを読み込んで処理を行い。。。といったようにバッチ処理が進行していきます。

全てのジョブ・ステップが完了した時点でジョブ全体は完了し、結果がスケジューラに通知されます。

xJCLにどのような情報を記述するのか、トランザクショナル・バッチの場合について紹介します。

先にお話したように、ジョブの内容は、ジョブ・ステップに分解できます。

各ジョブ・ステップでは、バッチ・ジョブ・ステップとバッチ・データ・ストリームのクラス名や、チェック・ポイント・アルゴリズムの指定、或いは、これらに付随した各種パラメータを記述します。



パラメータの変更があるたびに、xJCLを作成するとなると、似たようなxJCLを大量に管理しなくてはなりません。

或いは、例えば、ジョブに渡すファイル名に日付が入る場合など、そのたびにxJCLを変更することになりかねません。

そこで、xJCLをなるべく共用し、実行時に必要なパラメータだけを変更するための機能が、「代替プロパティ」です。

この図のように、$で始まる変数を記入することで、代替プロパティとして宣言したことになります。

その上で、各代替プロパティのデフォルト値を、<substitution-props>セクションに記載します。

代替プロパティは、実行時に値を指定することができます

例えば、ジョブ管理コンソールから、ジョブをサブミットする際に、「代替プロパティの変更」にチェックを入れると、

次の画面で、代替プロパティの値を変更することができます。

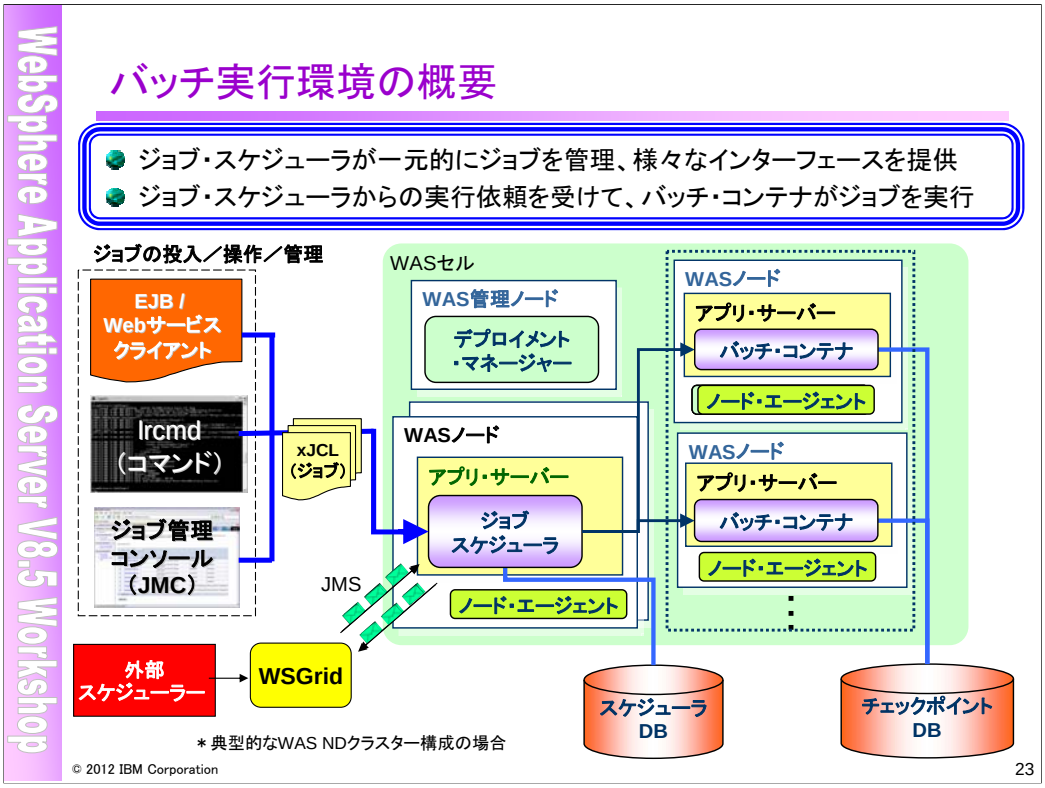
実は、この代替プロパティの機能は、先に紹介したParallel Job managerによるパラメータの分割でも利用されています。

WebSphere Application Server V8.5 Workshop

【参考】xJCLサンプル（一部）

```
<?xml version="1.0" encoding="UTF-8"?>
<job name="BatchSample" default-application-name="SampleBatchApp" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<jndi-name>ejb/com/ibm/ws/batch/SampleBatchAppBatchController</jndi-name>
   <step-scheduling-criteria>
      <scheduling-mode>sequential</scheduling-mode>
   </step-scheduling-criteria>
   <checkpoint-algorithm name="recordbased">
   <classname>com.ibm.wsspi.batch.checkpointalgorithms.recordbased</classname>
      <props>
         <prop name="recordcount" value="5" />
      </props>
   </checkpoint-algorithm>
   <results-algorithms>
      <results-algorithm name="jobsum">
         <classname>com.ibm.wsspi.batch.resultsalgorithms.jobsum</classname>
      </results-algorithm>
   </results-algorithms>
   <substitution-props>
      <prop name="sample.input.file" value="/tmp/indata.txt" />
      <prop name="sample.output.file" value="/tmp/outata.txt" />
   </substitution-props>
   <job-step name="Step1">
      <classname>com.ibm.websphere.batch.devframework.steps.technologyadapters.GenericXDBatchStep</classname>
         <checkpoint-algorithm-ref name="recordbased" />
         <results-ref name="jobsum" />
         <batch-data-streams>
            <bds>
               <logical-name>inputStream</logical-name>
               <props>
                  <prop name="IMPLCLASS" value="mysample.HelloBatchFileReader" />
                  <prop name="FILENAME" value="${sample.input.file}" />
               </props>
               <impl-class>com.ibm.websphere.batch.devframework.datastreams.patterns.TextFileReader</impl-class>
            </bds>
      。。。。。
</job>
```


xJCLの具体的なサンプルです。後半は割愛しています。



バッチ・ジョブ・ステップやバッチ・データストリームといったクラスを開発して、バッチ・コンテナーにデプロイし、ジョブ処理の内容を記載したxJCLを作成したら、いよいよバッチ・ジョブの実行です。

ジョブの実行、及び、制御のインターフェースは、3つあります。

1つは、ジョブ管理コンソールで、グラフィカルなWeb UIによる操作環境を提供します。

2つめは、コマンド・ラインツール、lrcmdです。xJCLや各種パラメータを指定して実行することで、サブミットができます。また、サブミット時のリターンに含まれるジョブIDを元に、色々な操作を行うことが可能です。

更に、このようなジョブ操作をプログラムから行いたい場合のために、EJB、或いは、Webサービスによる APIを提供しています。

これらのインターフェースについて共通する点として、ジョブのサブミットが成功した時点で、制御がクライアント側に戻る点です。

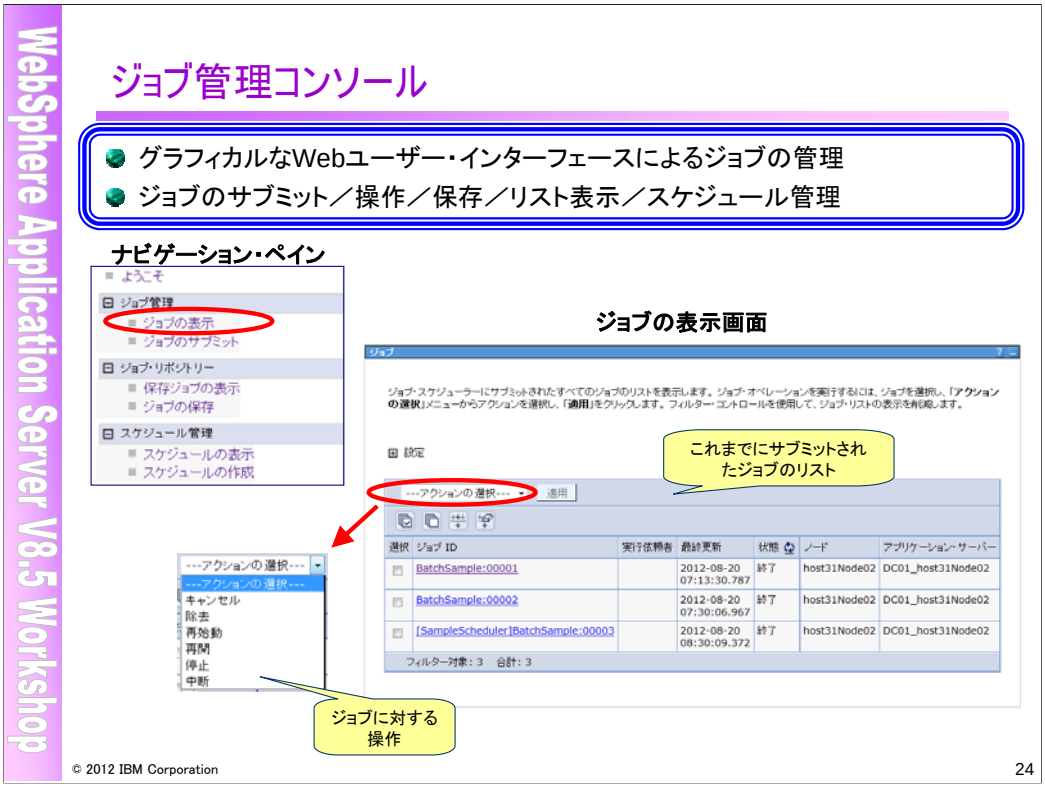
ジョブの実行には、数時間かかるケースもありますから、これは当然の動作です。

しかし、いわゆるエンタープライズ・スケジューラ製品、例えば、当社製品であればTivoli Workload Schedulerなどでは、実行が同期で行われることを前提としています。つまり、制御が戻った時点で、「ジョブが終了した」と見なして、次の実行を開始してしまいます。

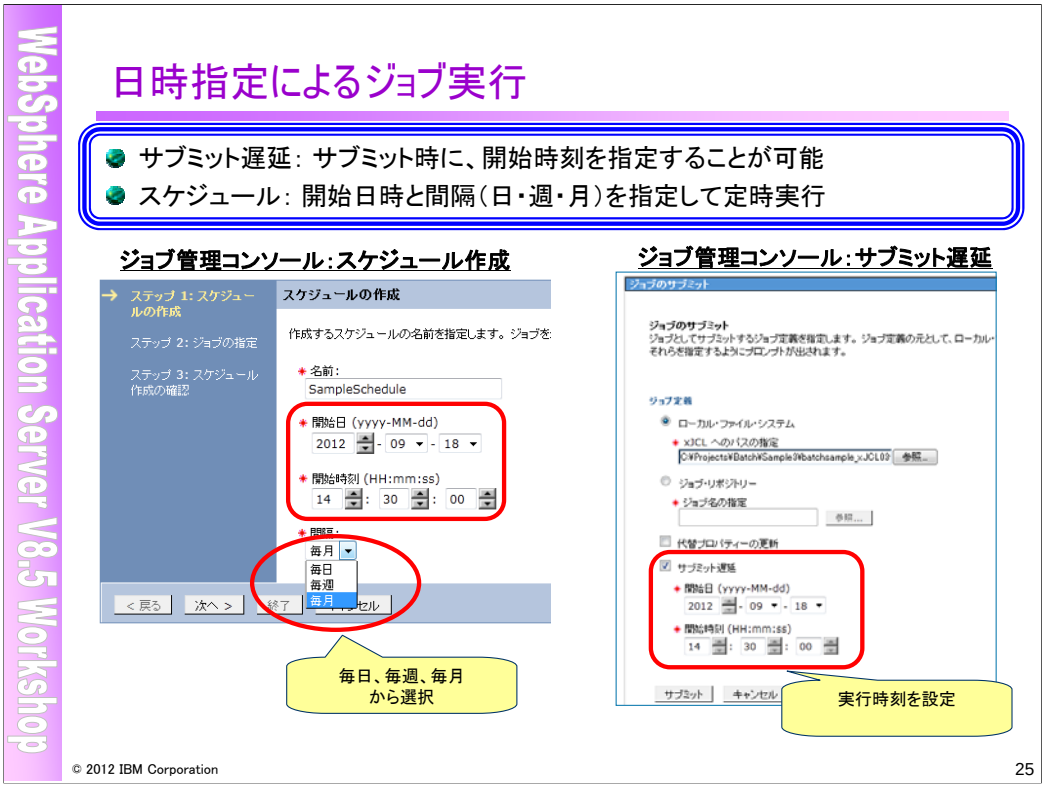
そのため、通常のスケジューラとのジョブ連携のためのユーティリティとしてWSGridが提供されています。外部のスケジューラは、このWSGridを通じて、WASのバッチと連携することができ、企業のジョブ・ネットに、WebSphereバッチを組み込むことが可能になります。WSGridの詳細は、後ほど説明します。

ジョブを実行するWAS側の構成ですが、まず、ジョブを一元的に管理するジョブ・スケジューラがあります。ジョブ・スケジューラは、ジョブ情報をDBに入れて管理しています。

ジョブ・スケジューラは、ジョブを受付けると、アプリケーション・サーバー上で稼動する、いずれかのバッチ・コンテナに、そのジョブの実行を依頼します。バッチ・コンテナーは、実際のジョブを実行し、例えば、それがトランザクショナル・バッチの場合は、一定のタイミングで取得したチェック・ポイント情報をDBに書き込みます。



ジョブ管理コンソールの、「ジョブの表示」の画面です。この画面では、ジョブのリストが、表示され、これらのジョブに関する各種操作を行うことができます。操作は、リストから選択できます。また、ジョブIDをクリックすると、ジョブ・ログが表示されます。ジョブ・ログについては、後ほど説明します。ジョブ管理コンソールで行える操作は、インターフェース、コマンド・ライン・ツールlrmcd、或いは、EJB及びWebサービスのAPIからも行うことができます。

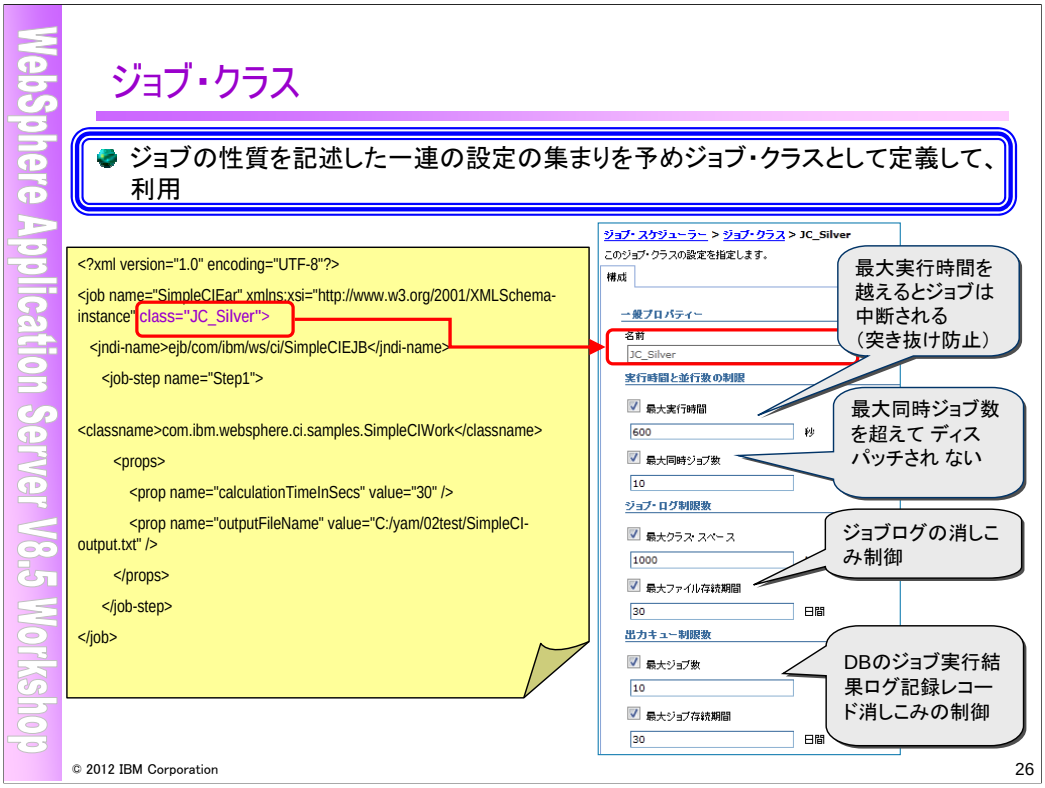


通常、ジョブがサブミットされると、実行可能なバッチ・コンテナがあれば、ジョブは即時に実行を開始します。

一方、ジョブの実行日時を指定したい場合があります。

例えば、今、ジョブをサブミットして、後で遅延で一度だけ実行したい場合です。このような遅延実行は、サブミット画面の「サブミット遅延」を選択し、時刻を指定してサブミットすることで行えます。

また、もう一つは、バッチ実行を繰り返し定期的に行いたい場合です。ジョブ実行の「スケジュール」を作成することで、1日、1週間、或いは、一ヶ月の周期で定期的にジョブ実行することができます。

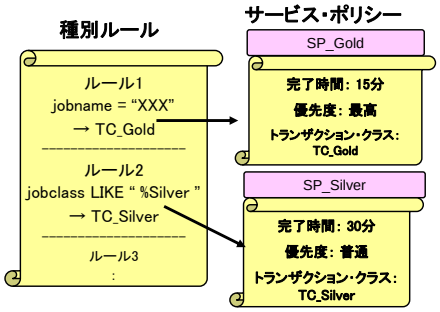


一連の性質が似通ったジョブをグルーピングするための機能として、メインフレームでお馴染みのジョブ・クラスを設定し利用することができます。最大実行時間や最大同時実行ジョブ数、後述するジョブ・ログの消去のタイミング等々をジョブ・クラスに対して設定できます。

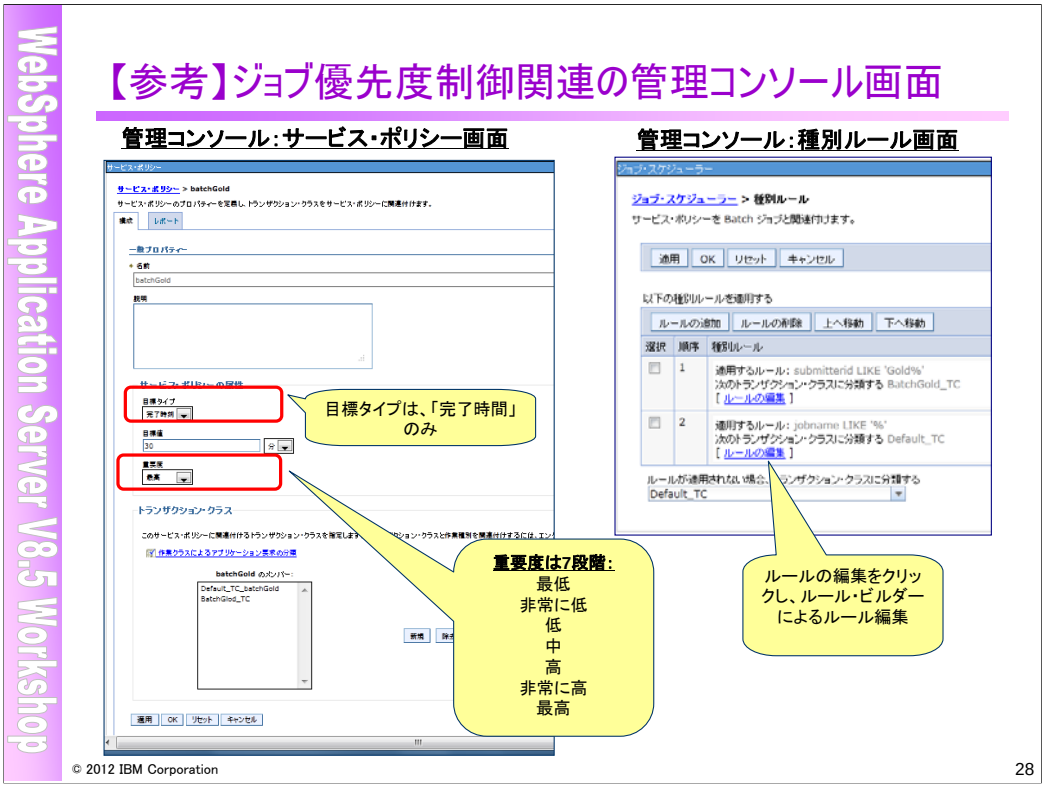
ジョブのxJCL中で、このジョブ・クラスを指定すれば、適切なスケジューリングのための情報をスケジューラに与えることができます。

ジョブ優先度制御 New

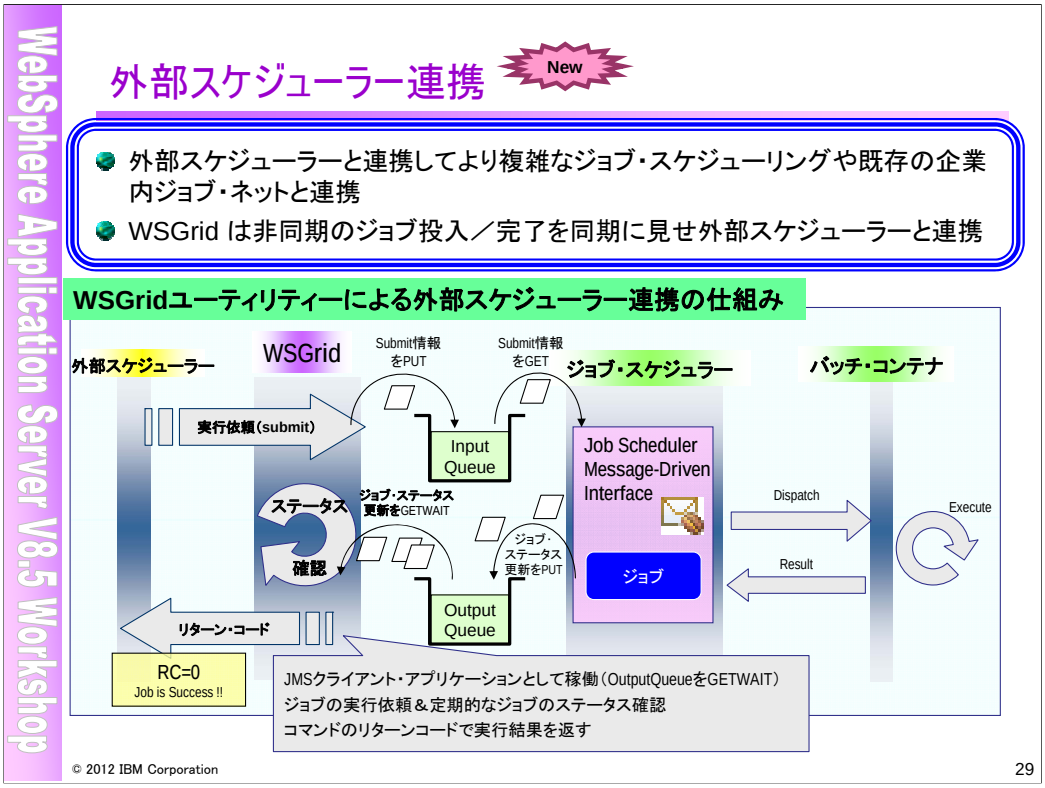
- **サービス・ポリシー**
  - 「目標完了時間」と「重要度」を設定
  - ジョブのディスパッチ先の選択、及び、実行順序の優先度に反映
  - 各ジョブとサービス・ポリシーとの紐付けは、種別ルールに記述
- **種別ルール**
  - ジョブ実行依頼者やジョブクラス、ジョブ名等のルールによって、各ジョブをトランザクション・クラスに対応付け
  - トランザクション・クラスに対応するサービス・ポリシーが適用される



ジョブのスケジューリングに対して、優先h度を設定することもできます。
優先度や目標完了時間といった情報をサービス・ポリシーとして定義します。
サブミットされたジョブがどのサービス・ポリシーに従って実行されるかは、種別ルールに記述します。
例えば、ジョブ名や、ジョブ・クラス、或いは、ジョブをサブミットしたユーザーIDなども指定することができます。



左側は、サービス・ポリシーの設定画面、右側が、種別ルールの設定画面になります。
これらの設定は、ジョブ管理コンソールではなく、WASの管理コンソールで行います。



先にお話したように、外部スケジューラと連携するために、ジョブ実行の完了を待って制御を戻すためのWSGridユーティリティが提供されています。

ここでは、その大まかな動作概要です。

WSGridは、実行依頼とともに起動されて、サブミット情報を送信側のJMSキューにPUTします。ジョブ・スケジューラは、その情報を取り出して、バッチ・コンテナに実行依頼します。この間、WSGridは、受信側のJMSキューをGET Waitしています。ジョブ・スケジューラからジョブ完了のメッセージを受け取った時点で、WSGridは実行を終了して外部スケジューラに制御を戻します。このときに、正常終了であれば、リターンコード0を返します。

ロギング機能

- ジョブログ
  - スケジューラー及びバッチ・コンテナが出力するジョブ実行に関する詳細なログ
  - ジョブ管理コンソールから参照及びダウンロードが可能
  - ジョブからの出力を制御できるジョブログ・フィルターSPI
  - 次のディレクトリー下2つのファイル（part.0.log＆part.1.log）に出力
    <WASプロファイル・ディレクトリー>/joblogs/<JOB_ID>/<yyyymmdd_HHMMSS>/
    （ yyyy: 西暦年／mm:月／dd：日／HH：時刻／MM：分／SS:秒）

New

- 使用状況の記録（チャージバック・アカウンティング機能）
  - 課金用の情報をDBに記録
  - ジョブ毎に、消費CPUを10^{-9}秒単位で記録
  - 直接SQLで情報を取り出すことも可能
  - zOS版では、SMFへの記録も可能

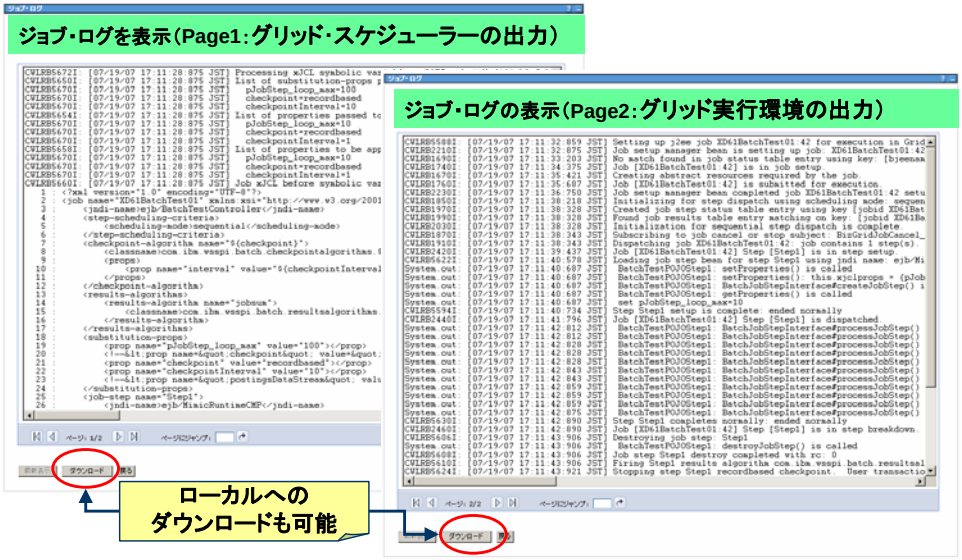
ログや記録に関する2つの機能を紹介します。

1つ目は、ジョブ・ログで、実行しているxJCLをはじめ、詳細の実行状況のログを出力します。ジョブ・ログの内容は、すでに述べたように、ジョブ管理コンソールから参照できます。また、必要に応じて、ジョブ管理コンソールの画面から、ジョブ・ログファイルをダウンロードすることも可能です。次ページに、この画面のキャプチャを添付してありますので、参照ください。

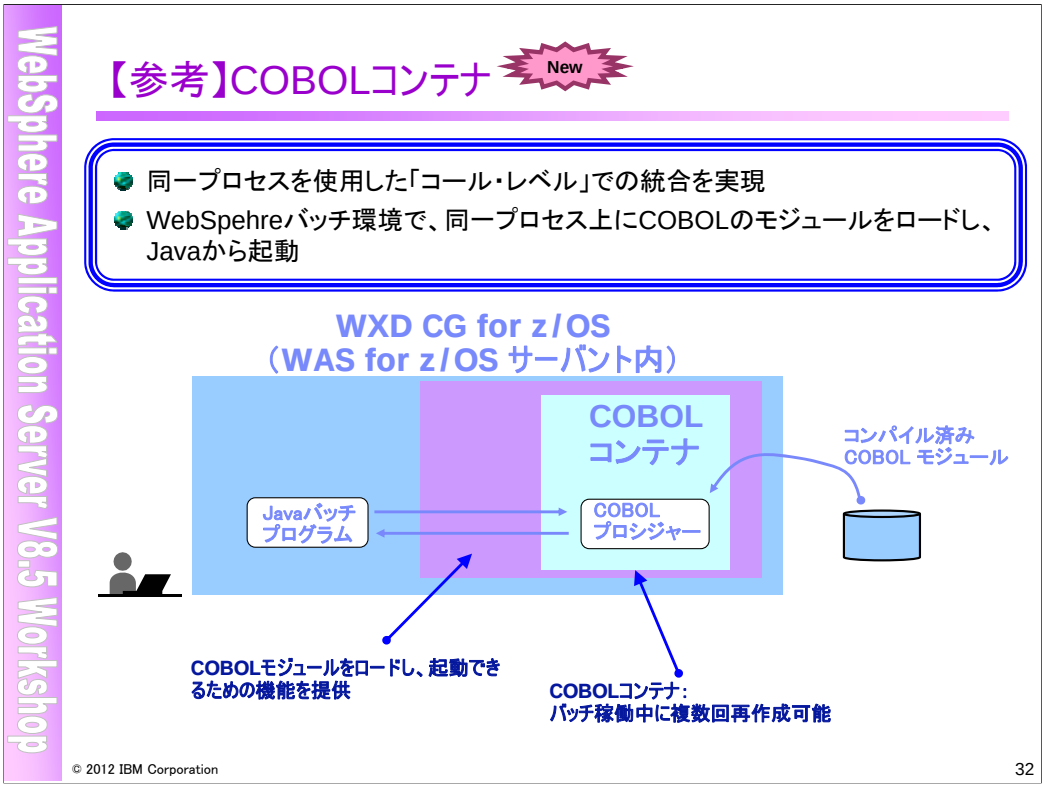
2つ目は、いわゆる課金のための、使用状況を記録するための、チャージバック・アカウント機能です。各ジョブで使用したCPU時間を10^{-9} 秒単位でDBに記録します。情報は、直接SQLで取り出すことも可能です。zOS版では、お馴染みのSMFへ記録することも可能です。

【参考】ジョブ管理コンソールでのジョブ・ログ参照

ジョブ管理コンソールから一括してジョブ・ログを確認可能

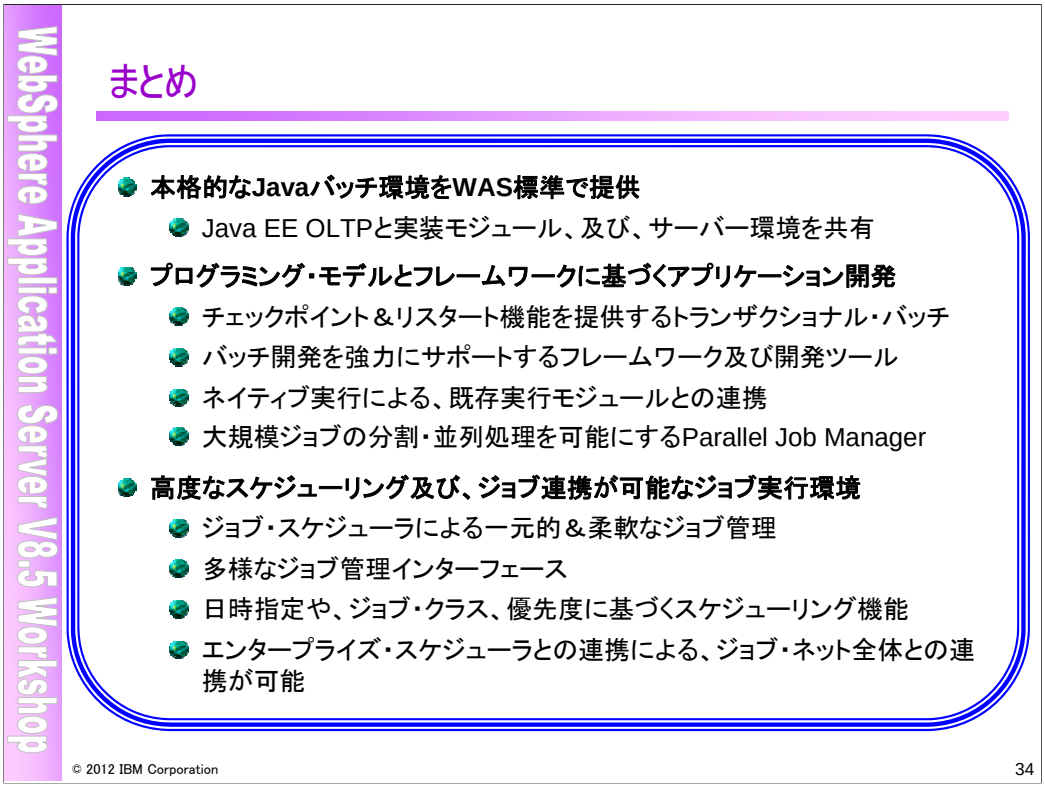


ジョブ管理コンソールから、ジョブログを参照している画面です。画面下の「ダウンロード」ボタンをクリックすることで、ダウンロードできます。



zOS版固有の機能として、COBOLのモジュールを、バッチコンテナと同じプロセス内で実行することが可能です。詳細は割愛します。

まとめ



2章でお話したように、WASのバッチ機能は、しっかりとしたプログラミング・モデルに基づき、チェック・ポイント&リスタートや分割・並列処理機能、及び、ネイティブ実行といった機能を提供しています。また、開発を強力にサポートするフレームワークを用意しています。

ランタイム環境では、信頼性の高いバッチ・コンテナに加えて、優先度や、ジョブ・クラスに基づくスケジューリング機能を備えています。

また、スケジューリング機能を補完するとともに、WASバッチを他環境のジョブと連携したジョブ・ネットに組み込むことができる、ジョブ連携機能も準備しています。

このように、WebSphereバッチは、企業内の本格的なバッチ処理を行うために必要十分と思われる機能を提供しています。

参考文献

- WebSphere ライブ WAS道場「知っていますか？ WASでのJavaバッチ処理」
  http://www.ibm.com/software/jp/websphere/events/livestream/was_ondemand.html#sec3
- DeveloperWorks記事「作ってみようJavaバッチ ～ WebSphere Application Serverのバッチ機能を使う」
  http://www.ibm.com/developerworks/jp/websphere/library/was/was_batch/
- WebSphere XD V6アップデート&デザインワークショップ資料：J2EEバッチシステムのつぼ
  http://www.ibm.com/developerworks/jp/websphere/library/wxd/wxd6_updatews/
- DeveloperWorks Japan: WebSphere XD Compute Grid
  http://www.ibm.com/developerworks/jp/websphere/category/wxd/cg.html
- WebSphere Application Server V8.5 InfoCenter
  http://pic.dhe.ibm.com/infocenter/wxsinfo/v8r5/
- WebSphere XD Compute Grid V8 InfoCenter
  http://publib.boulder.ibm.com/infocenter/wasinfo/cgwas80/index.jsp