

Java EE 7 アプリケーション設計ガイド

JSR-352 Batch Applications for Java Platform 編

日本アイ・ビー・エム システムズ・エンジニアリング株式会社



Disclaimer

- この資料は日本アイ・ビー・エム株式会社ならびに日本アイ・ビー・エム システムズ・エンジニアリング株式会社の正式なレビューを受けておりません。
- 当資料は、資料内で説明されている製品の仕様を保証するものではありません。
- 資料の内容には正確を期するよう注意しておりますが、この資料の内容は2015年11月現在の情報であり、製品の新しいリリース、PTFなどによって動作、仕様が変わる可能性があるのでご注意ください。
- 今後国内で提供されるリリース情報は、対応する発表レターなどでご確認ください。
- I B M、I B Mロゴおよびibm.comは、世界の多くの国で登録されたInternational Business Machines Corporationの商標です。他の製品名およびサービス名等は、それぞれ I B Mまたは各社の商標である場合があります。現時点での I B Mの商標リストについては、www.ibm.com/legal/copytrade.shtmlをご覧ください。
- 当資料をコピー等で複製することは、日本アイ・ビー・エム株式会社ならびに日本アイ・ビー・エム システムズ・エンジニアリング株式会社の承諾なしではできません。
- 当資料に記載された製品名または会社名はそれぞれの各社の商標または登録商標です。
- JavaおよびすべてのJava関連の商標およびロゴは Oracleやその関連会社の米国およびその他の国における商標または登録商標です。
- Microsoft, Windows および Windowsロゴは、Microsoft Corporationの米国およびその他の国における商標です。
- Linuxは、Linus Torvaldsの米国およびその他の国における登録商標です。
- UNIXはThe Open Groupの米国およびその他の国における登録商標です。

目次

1. はじめに

- 1.1. バッチ処理 について
- 1.2. Javaバッチ登場の背景と現状の課題
- 1.3. JSR-352におけるバッチ処理

2. 構成要素

- 2.1. Job
- 2.2. Step
- 2.3. その他コンポーネント

3. JSL (Job Specification Language)

- 3.1. JSLとは？
- 3.2. その前に……Listener と Properties
- 3.3. JSLでの変数使用

4. 各構成要素の説明と設定方法

- 4.1. Job
- 4.2. Step
- 4.3. Jobの処理順序制御
- 4.4. 実行要素
- 4.5. ジョブおよびステップのステータス

5. ジョブの管理・実装

- 5.1. JSR-352におけるバッチ処理のランタイム
- 5.2. ジョブの管理
- 5.3. ジョブの実装

6. サンプル・アプリケーション

- 6.1. 開発環境について
- 6.2. 開発の流れ
- 6.3. 手順解説

本文書の位置づけ

- この文書は、Java EE 7 アプリケーション設計ガイド シリーズにおいて、「JSR-352 Batch Applications for Java Platform」を対象に主に仕様解説の観点で記述したものです。
- 本文書の6章において、WAS Libertyをランタイムとして、サンプル・アプリケーションの開発から実装までを紹介しています。
- 今後、この文書の記載内容をベースに、WAS Libertyでの実装例や役立つTipsなどをトピック毎にまとめ、developerworksにて随時リリースする予定です。
 - <http://www.ibm.com/developerworks/jp/websphere/category/was/>

1. はじめに

1.1. バッチ処理 について

1.2. Javaバッチ登場の背景と現状の課題

1.3. JSR-352におけるバッチ処理

1.1. バッチ処理について

バッチ処理とは？

- データを纏めて一括処理を行う処理方式
- 複数の手順からなる処理を自動的に連続して処理を行う処理方式
- 処理中にユーザーとの対話は発生しない

そもそもなぜバッチ処理は必要か？

- 制約から派生する必要性
 - 業務上の「締め」という概念
 - 外部業務との連携上の制約
- 処理性能向上からの必要性
 - 一括処理によりオーバーヘッド削減
 - リソースの有効活用（即時性が求められないものをバッチ処理）

1.2. Javaバッチ登場の背景と現状の課題

Javaバッチ登場の背景と現状の課題

- 企業内には、オンライン処理以外にも多様なワークロード の形態 が存在
 - 実行するタイミングが決まっている定例処理（日次処理、月次処理、期次処理等）
 - スループットが要求される一方で、即時性は要求されない処理
 - （例：売り上げ集計、カード利用料金引き落とし、預金金利決算など）
 - 膨大なCPU時間を要する金融シミュレーションなどの計算集約型処理
- Webアプリケーション環境は、オンライン処理以外 が苦手（だった）
 - クライアント・サーバー型オンライン処理（1リクエスト1応答1同期点処理）
 - 長時間処理に向かない（応答タイムアウトがある）
- 分散系との混在による運用コストの増大
- ホスト系技術者の減少

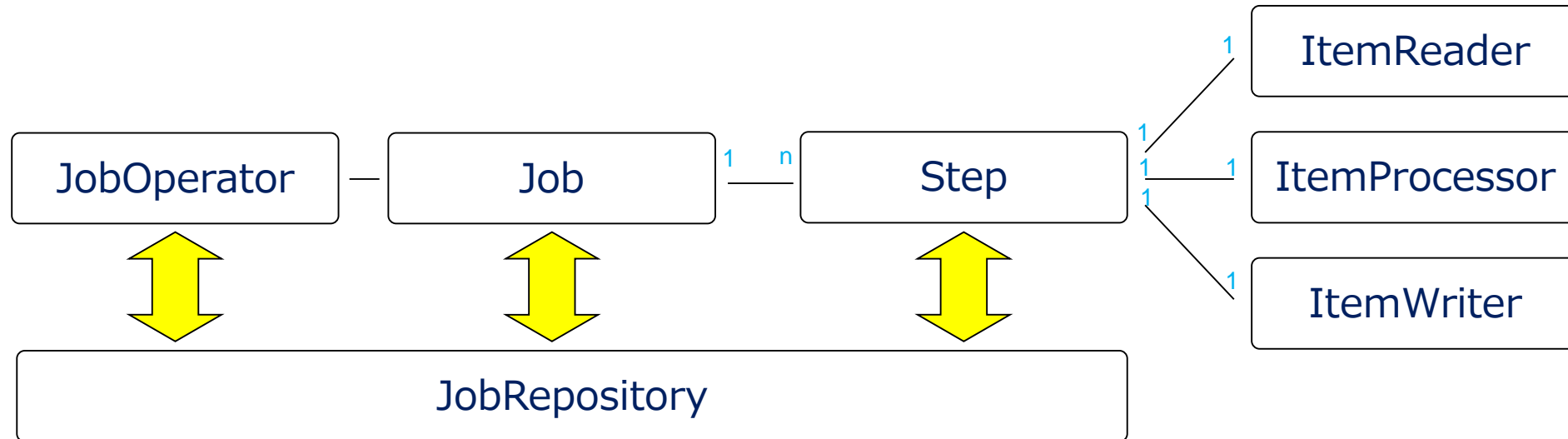
1.2. Javaバッチ登場の背景と現状の課題

「独自フレームワークの群雄割拠と **Java EEによる標準化**」

- Javaによるバッチ処理実行のために多くの製品やOSSが提供された
 - IBMからWCG (WebSphere Compute Grid)
 - OSSとしてはSpring Batchなど
 - 各社の内製フレームワーク
- 標準化の要請
 - ソフトウェア資産の保護やスキルの共通化
 - オンライン処理と利用技術の共通化
- Java EE 7の策定にあたって新仕様として追加
 - JSR 352 Batch Applications for the Java Platformとして標準化

1.3. JSR352におけるバッチ処理

- 以下は、典型的なバッチ処理のアーキテクチャーです。
 - ジョブは複数のステップから構成されます。
 - ステップは1つのItemReader、ItemProcessor、ItemWriterを持ちます。
 - ジョブは、JobOperator によって起動されます。
 - ジョブの実行状況は、JobRepositoryに格納されます。
- 次章にて、これらのコンポーネントの詳細について説明します。



2. 構成要素

ここでは、JSR-352でのバッチ処理を概観し、主要な構成要素とその役割を紹介します。

2.1. Job

2.2. Step

2.3. その他コンポーネント

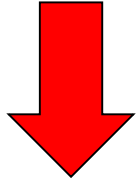
2.1. Job

Job



「**Job**」は、バッチ全体のプロセスをまとめたエンティティです。Jobは、1つあるいは複数の「**Step**」をまとめたもので、「Step」の制御情報や、各「Step」に共通のプロパティ設定などを行います。Jobには、再起動処理の可否も設定します。

例えば・・・日次のジョブ

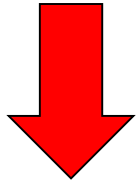


JobInstance



「**JobInstance**」は、Jobの実行単位に相当します。例えば日次でのジョブの場合、2015/10/06 のジョブ、などに相当します。

例えば・・・2015/10/06 のジョブ



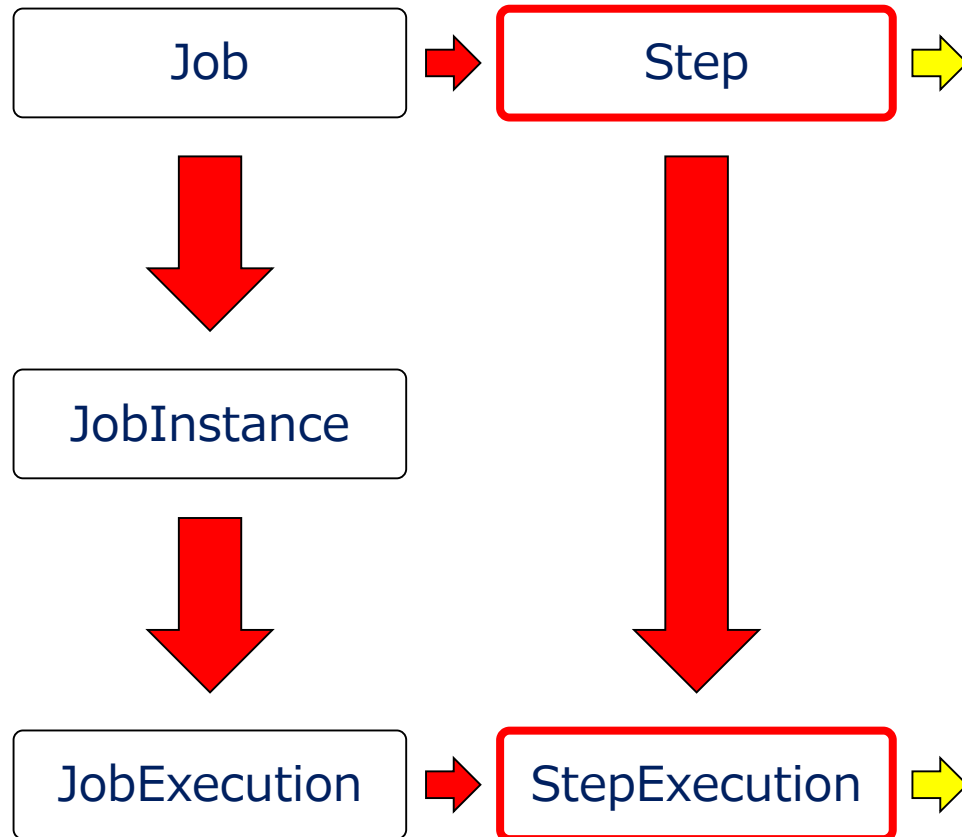
JobExecution



「**JobExecution**」Job実行の1試行に相当します。再処理などで実行した場合、それは同じJobInstanceの異なるJobExecution、ということになります。また、ジョブの起動・再起動時には、「**JobParameter**」を指定することができます。

例えば・・・2015/10/06 のジョブ、1回目の試行

2.2. Step

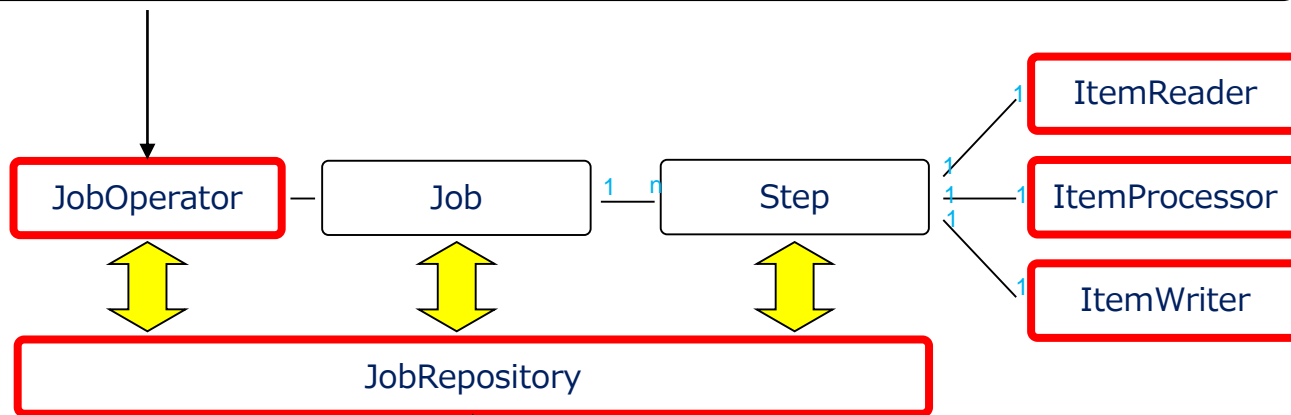


「**Step**」は、ジョブを構成する処理をまとめたオブジェクトです。よって、ジョブは1つ以上のステップで構成されることになります。バッチ処理の実行・制御に必要な情報を含んでいます。

「**StepExecution**」は、Stepの実行の1試行に相当します。StepExecutionは、Stepが実行された際に実際に生成されます。前Stepの実行が失敗してStepが実行されなかった、などの場合は生成されません。

2.3. その他コンポーネント

「**JobOperator**」は、ジョブの処理に関連する、起動、停止などの操作コマンドや、リポジトリの操作など、全ての管理的なインターフェースとなるコンポーネントです。



「**ItemReader**」は、Stepへの入力となるデータの取得を担います。入力データが枯渇した場合はその旨をStepへ伝えます。

「**ItemProcessor**」は、データの処理を担います。ItemReaderが取得したデータの変換やビジネスロジックの実行を行い、出力をItemWriterに渡します。

「**ItemWriter**」は、Stepからのデータの出力を扱います。次に行う処理などには関与せず、渡されたデータのみを扱います。

「**JobRepository**」は、実行中、および過去に処理したジョブの履歴を保持するコンポーネントです。JobOperatorを介してリポジトリの操作を行います。JobRepositoryには、JobInstance、JobExecution、StepExecutionが含まれます。

3. JSL (Job Specification Language)

JSLはジョブ、ステップ、およびそれらの実行を規定します。

JSR-352では、JSLをXMLで記述します。よって、JSLは"Job XML"と呼ばれることもあります。

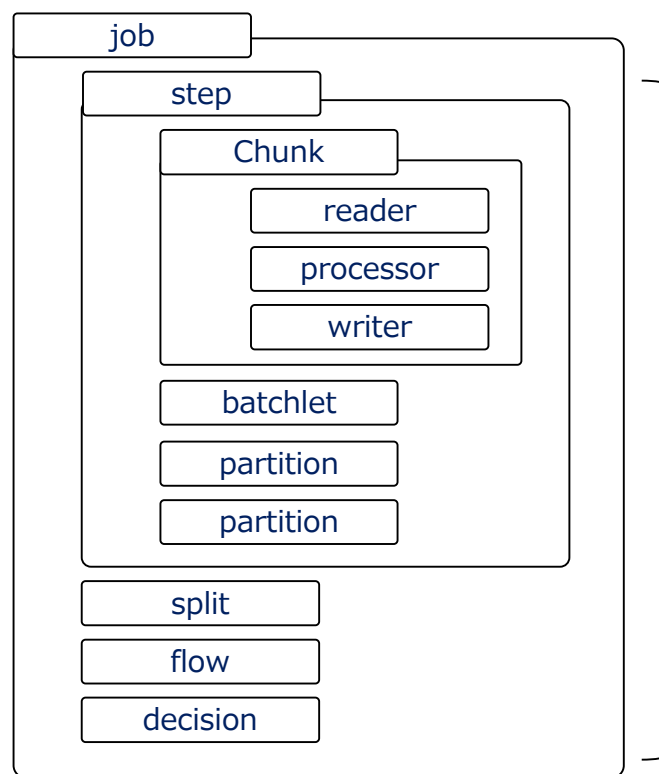
3.1. JSLとは？

3.2. その前に・・・Listener と Properties

3.3. JSLでの変数使用

3.1. JSLとは？

1. JSL (**J**ob **S**pecification **L**anguage) は、ジョブおよびステップの設定、およびジョブの処理内容の制御を規定するためのXMLファイルです。
2. 各設定要素の包含関係は大まかには以下のような構造となっています。次章で各構成要素の設定・構成の詳細を解説します。



実行要素
(Execution elements)

その他、JSR-352 では以下の説明があります。これらについても、構成要素の説明の中に散りばめて説明します。

- 遷移要素 (Transition elements)
 - [4.3. Jobの処理順序制御 \(1/2\)](#)
- バッチ・ステータス、Exitステータス
 - [4.5. ジョブおよびステップのステータス \(1/3\) - バッチ・ステータス、Exitステータスとは？](#)
- Job XMLでの変数使用
 - [3.3. Job XMLでの変数使用](#)

以降、エレメントと属性を以下のように現すこととします。



3.2. その前に・・・Listener と Properties

1. Listener と Properties

- これらは、Job XML内の各設定箇所で度々出てきますので、先立って説明します。
- Listenerは、それぞれの設定項目の範囲で個別に設定が可能なもので、カスタムで自作したコードを呼び出すための拡張として用いる事ができます。
- Propertiesは、これらのListenerとして定義されたアーティファクトに、プロパティーとして渡す情報を定義するために使用されます。

2. 役割

- Listener
 - ステップ実行の様々なフェーズをインターセプトするリスナー・アーティファクトを定義します。
chunk型のStepでは、リスナーのバッチ・アーティファクトは以下のインターフェースを実装します。
 - StepListener, ItemReadListener, ItemProcessListener, ItemWriteListener, ChunkListener, RetryReadListener, RetryProcessListener, RetryWriteListener, SkipReadListener, SkipProcessListener, SkipWriteListener.
- Properties
 - バッチ・アーティファクトがバッチ・コンテキスト・オブジェクトを使用して取得できる、プロパティーの組み合わせを定義します。



3.3. Job XMLでの変数使用

■ 置換のルール

- Job XML内では変数の使用が可能です。
- 変数の解決は、ジョブの始動前に行われ、以下の順に変数値の設定の検索・置換が行われます。

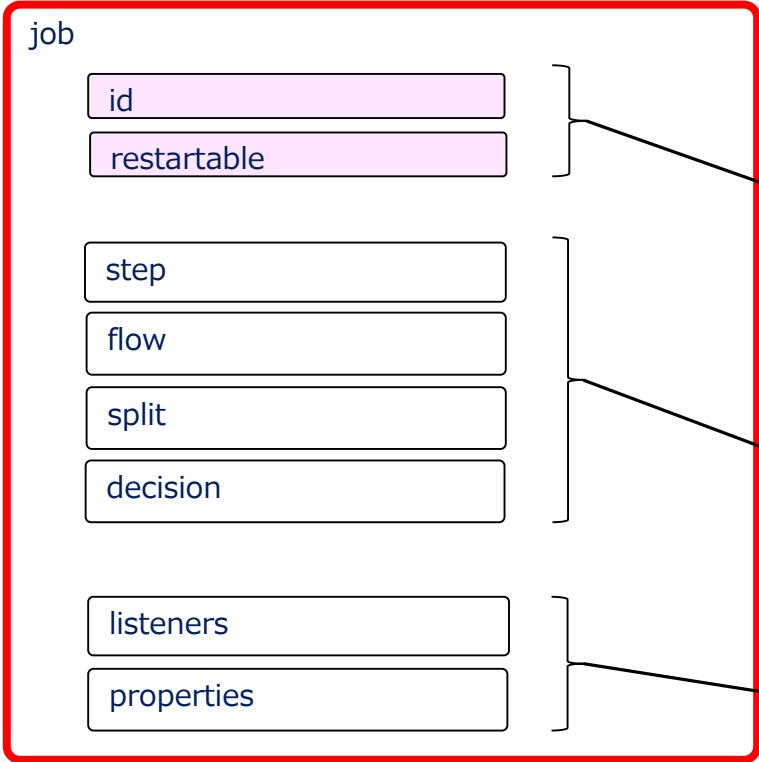
- jobParameters
 - ジョブ起動時に渡されるジョブ・パラメーター（実際の使用イメージはP80参照）でパラメーター/値が設定されます。
- jobProperties
 - JSL内で指定されているパラメーターに対して値の設定・検索がなされます。
- systemProperties
 - システム・プロパティーとして定義されているパラメーターに対して値が検索されます。
- partitionPlan
 - パーティション・マッパーによって返されるパーティション・プラン内で有効なパラメーターが返されます。
 - これは、定義されているパーティション・スコープ内でのみ有効です。

4. 各構成要素の説明と設定方法

ここでは、各構成要素の設定項目、およびその方法について解説します。
JSL内での設定項目と共に、その設定項目の機能解説をしています。

- 4.1. job
- 4.2. step
- 4.3. ジョブの処理順序制御
- 4.4. 実行要素
- 4.5. ジョブおよびステップのステータス

4.1. job - 設定項目



1. ジョブ定義は、最も粒度の大きい要素になります。

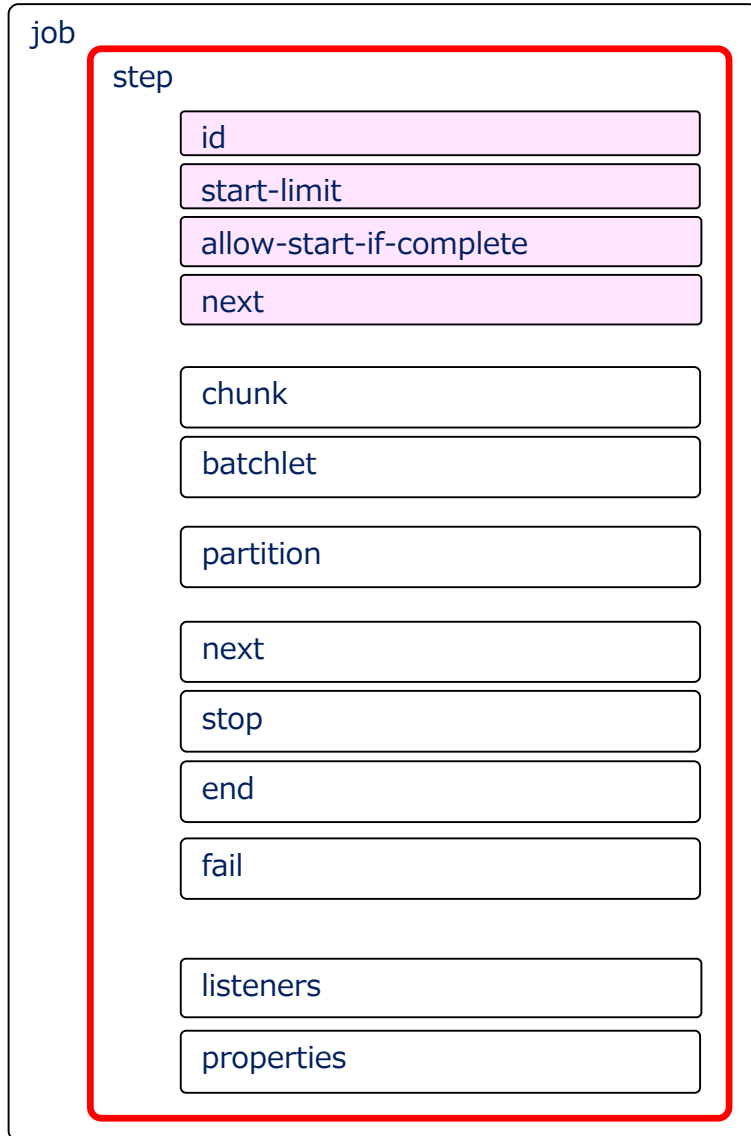
項目	説明
id	ジョブの論理名を指定します。ジョブの識別に使用されます。XMLのString属性の規約に従う必要があります。必須項目です。
restartable	このジョブが再始動可能か否かを true/false で指定します。オプションです。デフォルトは true です。

2. Step、Flow、Split、Decision の実行要素を子要素として持ちます。（詳細については後述）

3. listeners、propertiesの定義が可能

- 1. P16 で説明した listeners で処理をインターセプトして実行するアーティファクトの定義が可能です。
- 2. ジョブ・レベルのプロパティでは、ジョブ・コンテキスト全体で取得可能なプロパティの設定が可能です。

4.2. step - stepの属性情報、実行タイプ



1. ステップは、ジョブ実行時の最小単位です。属性として以下を持ちます

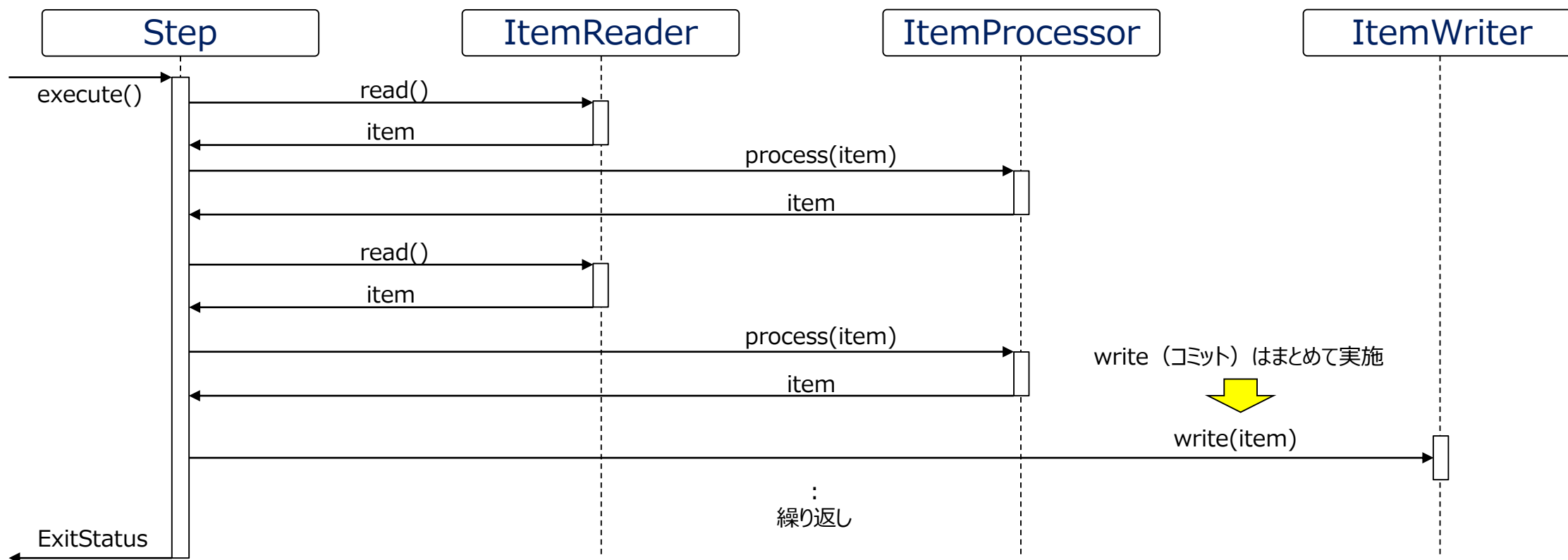
項目	説明
id	ステップの論理名を指定します。ジョブの識別に使用されます。XMLのString属性の規約に従う必要があります。必須項目です。
start-limit	このステップの起動、再起動で許容される試行回数を指定します。デフォルトは0で、無制限を意味します。この回数以上の試行がなされた場合はステップはFAILEDで終了します。
allow-start-if-complete	ジョブの再実行時に、（例え前回の試行が正常終了していたとしても）このステップの再処理を行うか否かを true / false で指定します。true の場合実行します。デフォルトは false です。
next	このステップの終了後に行う次の実行要素を指定します。step、flow、split、decision のいずれかを指定します。Loopが発生するような定義はできません。

2. ステップの実行タイプは、以下の2種類があります。

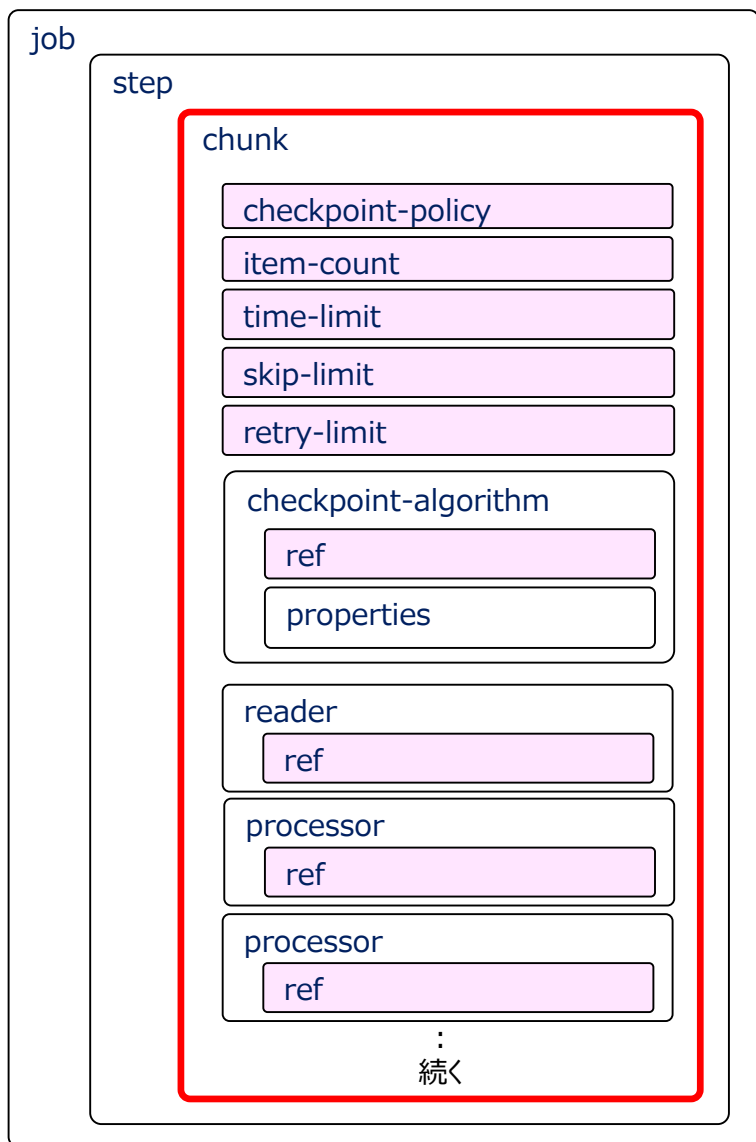
1. 「chunk（Chunk指向型処理）」
2. 「batchlet（タスク指向型処理）」

4.2. step - chunk (Chunk指向型処理) の詳細 (1/2)

- JSR-352では、バッチの処理形態として、「chunk (chunk指向型処理)」と「batchlet (タスク指向型処理)」を扱います。中でも、chunkパターンを基本の処理としています。chunkパターンでは、データを1回につき1 Item取得 (ItemReader) し、処理 (ItemProcessor) したうえでchunk (データの塊) として出力 (ItemWriter) します。出力は、指定した条件 (データの個数もしくは時間) に合致しときにまとめてコミットされます。
- 以下は、この振る舞いをシーケンスで模式的に表したものです。



4.2. step - chunk (Chunk指向型処理) の詳細 (2/2)



1. chunkは 属性として以下を持ちます。

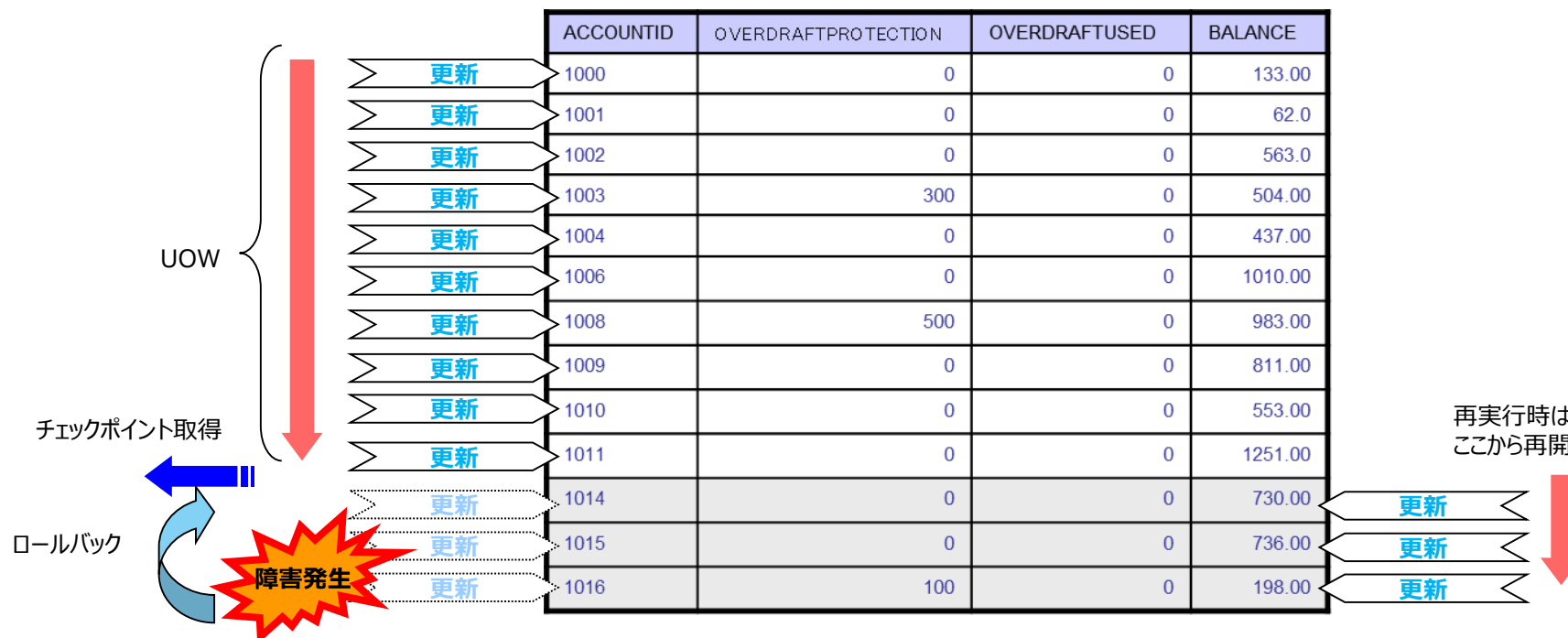
項目	説明
checkpoint-policy	コミットの振る舞いを規定するチェックポイントのポリシーを"item"もしくは"commit"で指定します。itemを指定した場合は一定数の処理後にチェックポイントを取得します。custom指定の場合は、別途指定するcheckpoint-algorithmの実装に応じてチェックポイントを取得します。デフォルトはitemです。
item-count	ポリシーがitemの場合に、チェックポイントを取得するchunkに含まれるitem数を指定します。デフォルトは10です。
time-limit	ポリシーがitemの場合に、チェックポイントを取得する間隔を秒数で指定します。デフォルトは0で、無制限（すなわち、時間間隔では取得しない）です。item-countとtime-limitの両方が指定されている場合は、最初にいずれかの条件に合致した時点でチェックポイントが取得されます。
skip-limit	スキップ対象として定義した例外が、何回発生したらステップをスキップするか、という回数の定義を行います。デフォルト無制限です。
retry-limit	リトライ対象として定義した例外が、何回発生したらステップを処理をリトライするか、という回数の定義を行います。デフォルト無制限です。

2. これらには、作成したアーティファクトを指定します。

項目	説明
ref	それぞれの項目に対応するアーティファクトを指定します。

【参考】チェックポイント

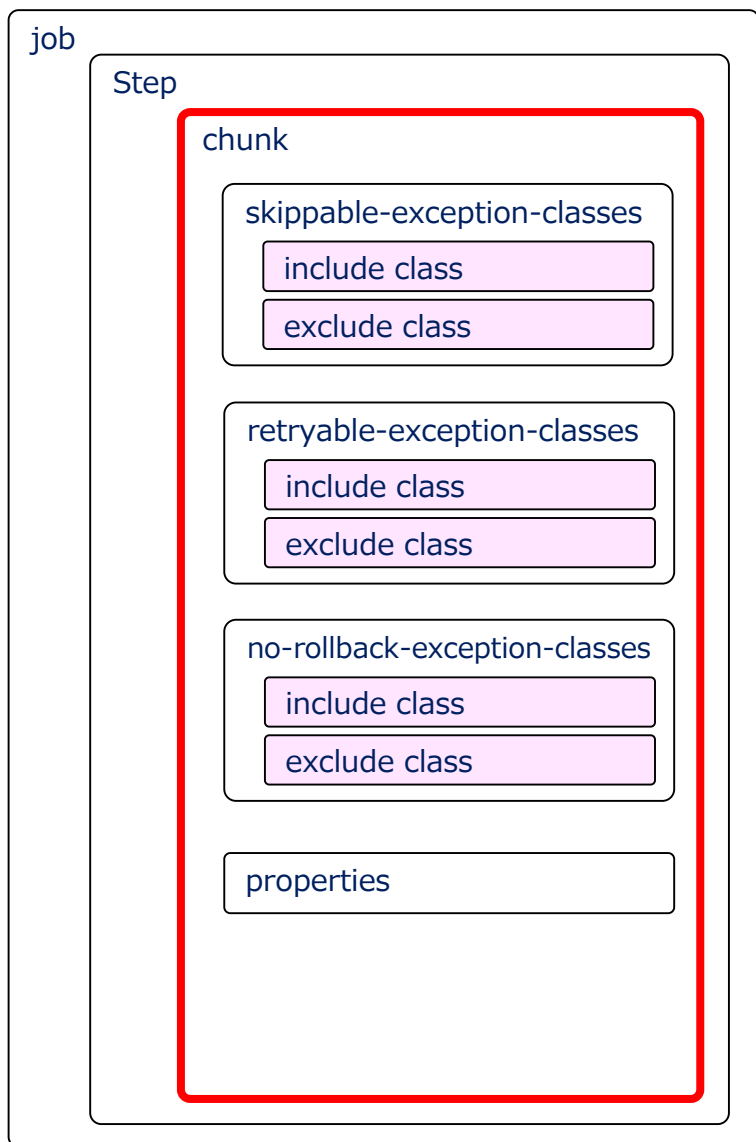
- チェックポイント機能とは、バッチが障害などの理由で途中停止した場合に備えて指定タイミングで定期的にコミットを取り、その進捗を記録することで、回復時に再実行可能な状態にしておく機能です。リスタート機能はチェックポイントの情報を読み取り、再実行された際に最後にコミットしたレコードの次の行からの処理を行う機能です。
- チェックポイント取得のインターバル、頻度は、システム全体のパフォーマンスに大きく影響するため、チューニングは非常に重要です。



4.2. step - chunk (chunk指向型処理) の詳細 - 例外発生時の対応(1/2)

- ステップ実行中に**例外が発生した場合**は・・・デフォルトでは、**Exitステータス : FAILEDで終了**します。
- しかし・・・reader / processor / writer にたいして、予め特定の例外を定義しておくことで、スキップ / リトライを設定できます。
 - **スキップ**
 - chunkの子エレメントとして、スキップ対象とする例外クラスを指定
 - **リトライ**
 - 同じくchunkの子エレメントとして、リトライ対象とする例外クラスを指定
 - デフォルトのリトライの振る舞いとしては、chunkを**ロールバック**して、（デフォルトの）チェックポイント・アルゴリズムに基づいてitem-count1から再開
 - オプションのchunk Listenerがstepに対して設定されている場合は、ロールバックの前にonErrorメソッドが呼ばれる
 - リトライ時に**ロールバックしない方法**も指定可能
 - retryable-exception-classes、no-rollback-exception-classes、両方に同一の例外が指定がされている場合、その例外発生時にロールバックはされません。
- 但し.....
 - 同じ例外がスキップ/リトライの両方で指定されていた場合は、リトライが優先されます。
 - リトライの実行中に、リトライ発生のトリガとなった例外と同じ例外が再度発生した場合、そのステップはスキップされます。
 - 設定方法については次ページを参照ください。

4.2. step - chunk (chunk指向型処理) の詳細 - 例外発生時の対応(2/2)



- chunkの例外発生時対応の元素として以下を持ちます。
 - **skippable-exception-classes**
 - 前頁記述のとおり、スキップ対象とする例外、スキップ対象としない例外を指定します。
 - **retryable-exception-classes**
 - 前頁記述のとおり、リトライ対象とする例外、リトライ対象としない例外を指定します。
 - **no-rollback-exception-classes**
 - リトライ対象として指定されている例外のうち、（デフォルトの振る舞いであるロールバック後のリトライをオーバーライドして）ロールバックせずにリトライを行う対象の例外、対象外の例外を指定します。
- それぞれのクラス指定では、以下の属性を持ちます。

項目	説明
include class	対象とする例外クラスもしくは例外のスーパークラスを指定します。複数指定可能です。
exclude class	対象から除外する例外クラスもしくは例外のスーパークラスを指定します。複数指定可能です。

4.2. step - batchlet (batchlet指向型処理) の詳細



1. batchlet型は、chunk型と比較してよりシンプルな処理形態です。
2. chunk型のような読み込み、処理、書き込み、の一連の流れではなく、ファイル転送や印刷、コマンド実行、などのタスク型の処理に適しています。
3. batchletの設定は非常にシンプルで、実行対象となるアーティファクトを指定し、必要に応じてプロパティーの設定をするのみです。以下のような指定を行います。

項目	説明
ref	Batchletとして実行するアーティファクトを指定します。

4. batchletインターフェースの詳細は、「[5.3. ジョブの実装 - batchlet 方式ステップの実装](#)」を参照ください。

4.2. step - stepのパーティショニング（1/4）

- バッチステップは、**パーティション化により複数のインスタンスに処理を分割して並行処理**させる事ができます。
 - パーティションの分割について
 - パーティション化されたステップは、**複数のインスタンスで実行**されます。
 - スレッドおよびパーティションの数は、パーティション・マップからの呼び出し、もしくは、JSL内（partition下のplanエレメント）で指定します。
 - 各パーティションには、どの範囲のデータを扱えばよいのか、というパラメーターを渡す必要があります。
 - 各スレッドはそれぞれ独立したコピーのステップとして動作します。よって、チェックポイント取得なども各パーティションでそれぞれ独立して行われます。
 - パーティション・リデューサーで、各パーティションの結果を統合・協調させることができるので、どこかでエラーが発生した場合に全体をバックアウトをさせる、という様なことも可能です。
 - 各パーティションの処理結果の統合について
 - 各パーティションの実行結果は、全体のステップの結果としてどうだったか判定する必要があります。
 - PartitionCollectorとPartitionAnalyzer がこの結果判定を担います。

4.2. step - stepのパーティショニング (2/4)

■ stepのパーティショニング

– **Partition Mapper** (パーティション・マッパー)

- パーティション数とスレッド数を計算するプログラムをパーティション・マッパーとして定義できます。
- また、パーティション・マッパーは、各パーティションで参照するプロパティも指定できます。
- 機能的に重複しますので、パーティション・プランを設定した場合は、Partition Mapperは使用できません。

– **Partition Reducer** (パーティション・リデューサー)

- パーティション処理において、UOW (Unit Of Work) の分割を行う機能を提供します。
- このパーティション・マッパーを利用してパーティション・ステップのライフサイクルにプログラマ的に介入することが可能になります。

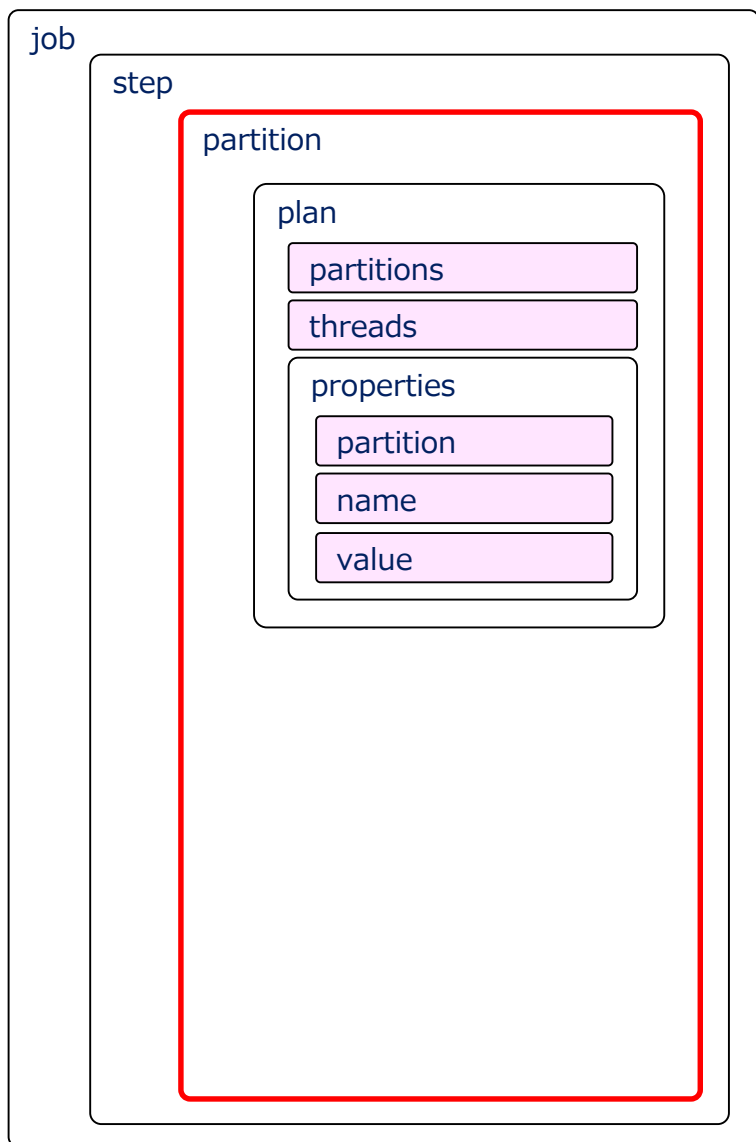
– **Partition Collector** (パーティション・コレクター)

- 各パーティションの実行結果をアナライザーに中継します。パーティションを実行するスレッド毎に別々のコレクターが稼動します。
- Chunk型ではチェックポイント取得時、およびステップ終了時に起動されます。batchlet型ではステップの終了時のみ起動されます。
- コレクターは、シリアライズされたjavaのオブジェクトをanalyzerに返します。

– **Partition Analyzer** (パーティション・アナライザー)

- ステップのメインのスレッドで稼動し、パーティション・コレクターから、各パーティションの実行結果を受信します。
- 各パーティションの処理終了後、各パーティションのExitステータスを受信します。
- アナライザーは、各パーティションの実行結果に応じてそのステップの実行結果をどう扱うか、のカスタムExitステータスの扱いを規定します。

4.2. step - stepのパーティショニング (3/4)



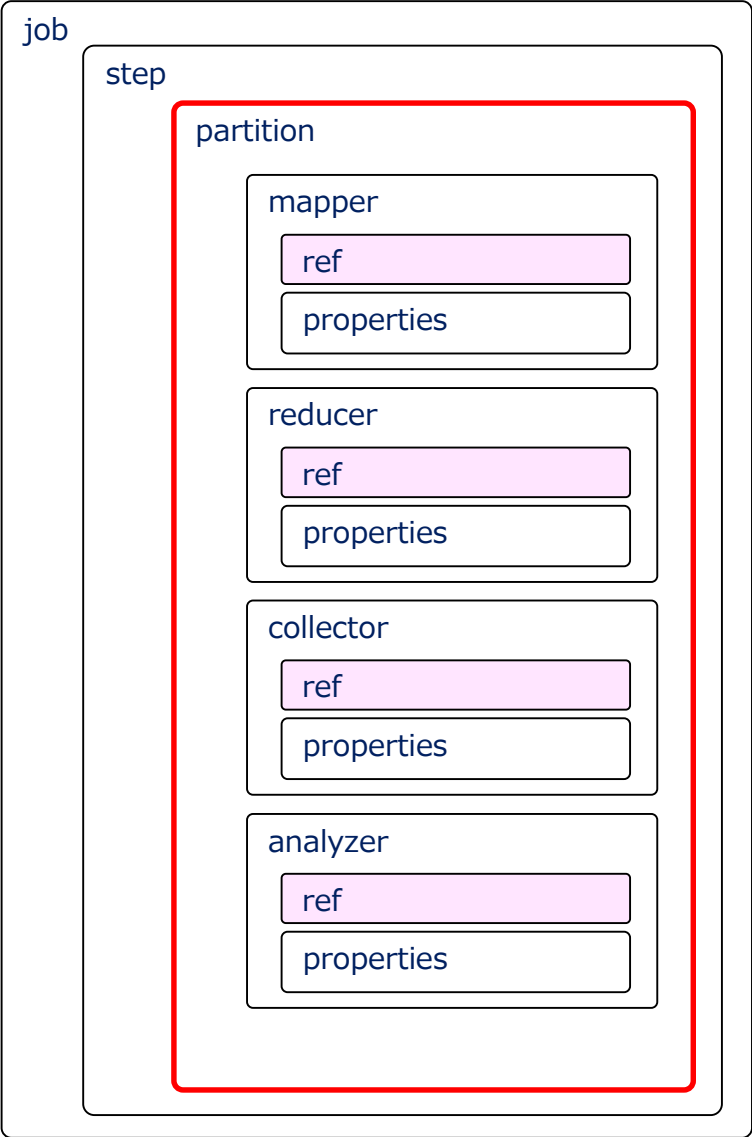
1. plan では、パーティション化されたstepの実行に関わる設定を行います。設定項目は以下のとおりです。

項目	説明
partitions	パーティション化されたステップのパーティションの数を指定します。オプションな属性で、デフォルトは1です。
threads	ステップが実行するスレッド数の最大値を指定します。JSR-352の仕様としては、要求された実行スレッド数でのステップ実行を保証するわけではない、としています。オプションな属性で、デフォルト値はpartitionsで指定されたパーティション数です。

2. また、各パーティションそれぞれに対してプロパティーを設定することができるようになっています。これを利用して、例えばパーティション毎に異なる処理対象ファイルを指定したり、処理対象のデータ範囲を指定したりすることができるようになります。

項目	説明
partition	プロパティーを適用する、0から始まるパーティション番号を指定します。
name	プロパティー名を指定します。
value	nameに対応するプロパティー値を指定します。

4.2. step - stepのパーティショニング (4/4)



1. [「4.2. step - stepのパーティショニング \(1/4\)」](#)で説明した、パーティション化された処理を担う各コンポーネントの設定は、以下のように行います。

項目	説明
ref	パーティション化されたステップのパーティションの数を指定します。オプションな属性で、デフォルトは1です。

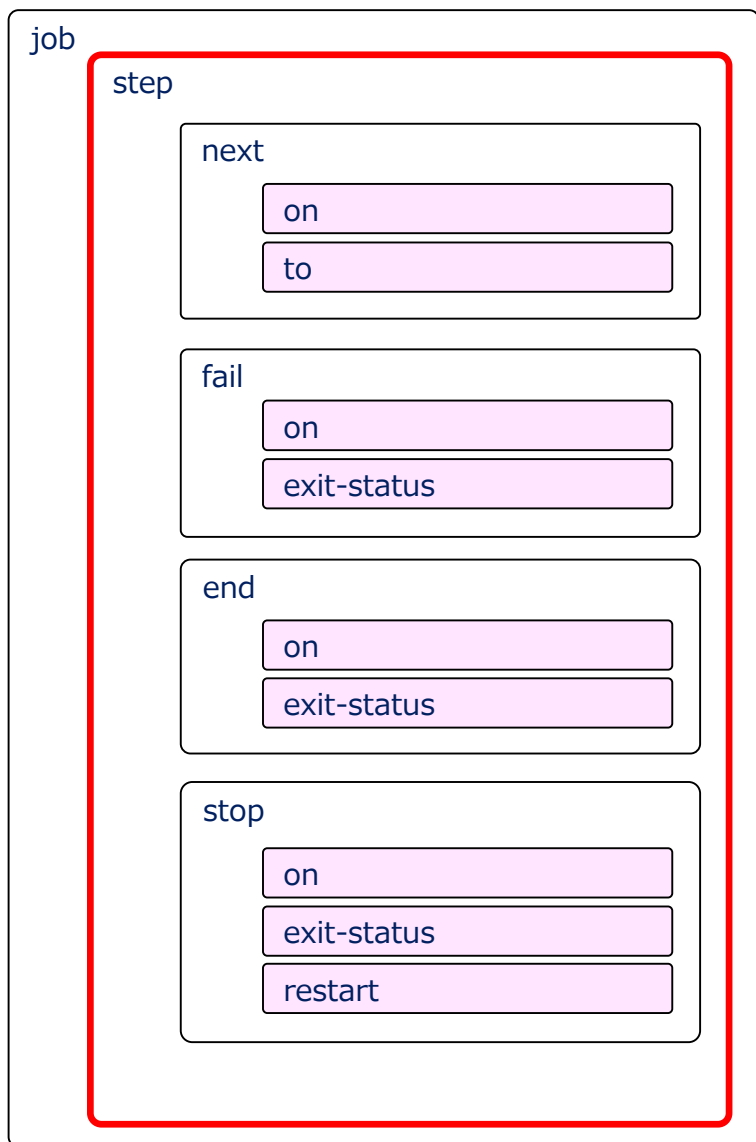
2. また、propertiesエレメントとして、各バッチ・アーティファクトに渡すプロパティ情報を以下の様に設定します。

項目	説明
name	各バッチ・アーティファクトに渡すプロパティ名を指定します。
value	各プロパティに対応する値を指定します。

4.3. Jobの処理順序制御 (1/2)

- 実行要素 (execution element) と遷移要素 (transition element)
 - ジョブの処理の流れは、実行要素のスコープ内で定義される遷移要素に応じて、次に実行する実行要素を指定するか、次のジョブ実行を終了させるかを定義することで行います。
 - つまり、実行要素の結果に応じて、次にこの実行要素を実行する、ジョブを終了させる、などの遷移要素の設定を行って制御することになります。
- 実行要素
 - step / flow / split / decision があります。詳細は次ページ以降で解説します。
- 遷移要素
 - 以下の要素があります。詳細は次ページで解説します。
 - **next** エlement
 - **fail** エlement
 - **end** エlement
 - **stop** エlement

4.3. Jobの処理順序制御 (2/2)

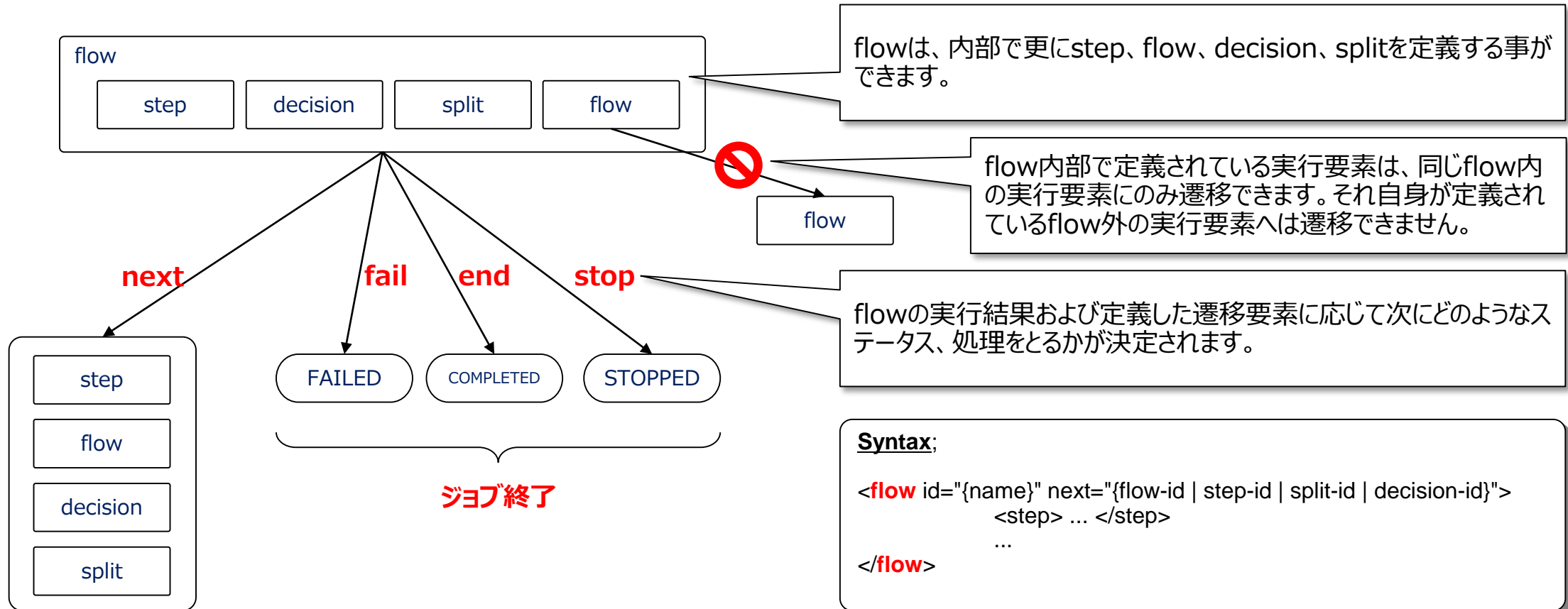


- **next** エLEMENT
 - 指定された、次の実行要素を実行します。
- **fail** エLEMENT
 - ジョブを、FAILEDのバッチ・ステータスで終了させます。
- **end** エLEMENT
 - ジョブを、COMPLETEDのバッチ・ステータスで終了させます。
- **stop** エLEMENT
 - ジョブを、STOPPEDのバッチ・ステータスで終了させます。
 - 併せて、ジョブがリスタートされた際に実行するジョブ・レベルでのstep、flow、splitを指定します。

項目	説明
on (必須)	(next, fail, end, stop で共通) Exitステータス値を指定します。"*" (0以上の文字列の合致) と、"?" (1文字の合致) を使用できます。
to (必須)	(next のみ) 適用された場合に次に遷移する実行要素のidを指定します。
exit-status (任意)	(fail, end, stop のみ) 指定した値でジョブに新しいExitステータスをセットします。指定しなかった場合は更新されません。
restart (任意)	(stop のみ) ジョブの再始動時に再始動の対象となるジョブ・レベルのstep、flow、splitを指定します。

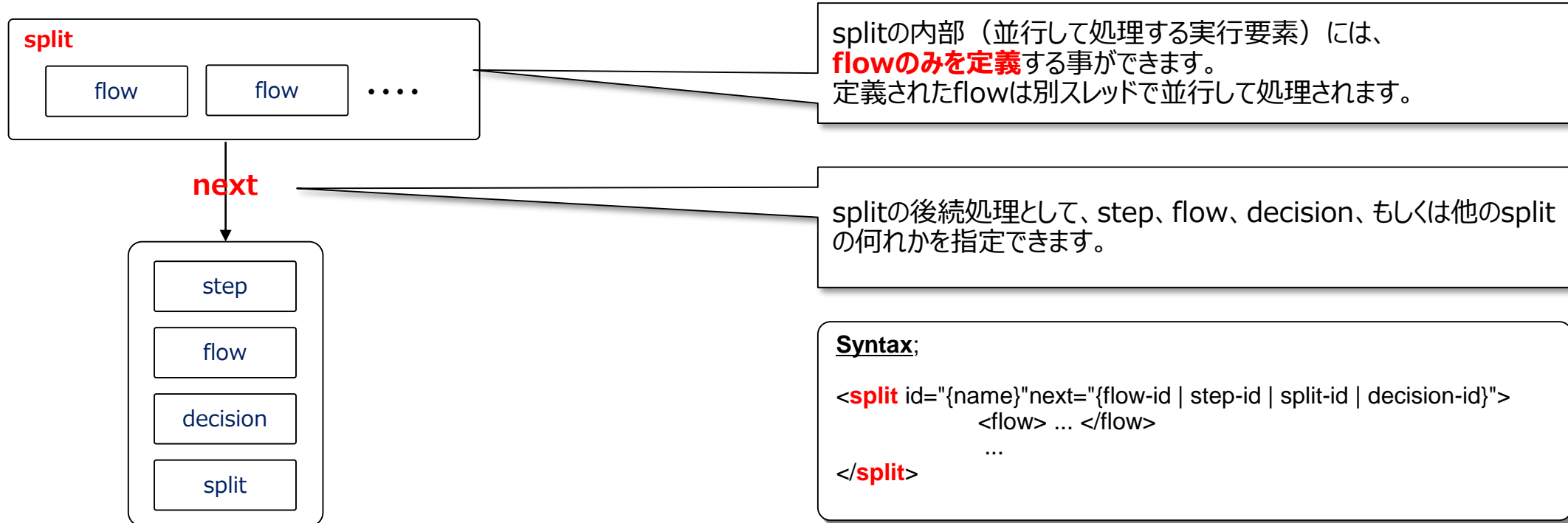
4.4. 実行要素 - flow

- flowは、1つの単位内で実行する、**実行要素の順序制御を定義**するものです。
- flowは、step、split、decision、もしくは別のflow に遷移できます。
- また、flowは、その処理単位内にstep、flow、decision、splitを定義する事ができます。



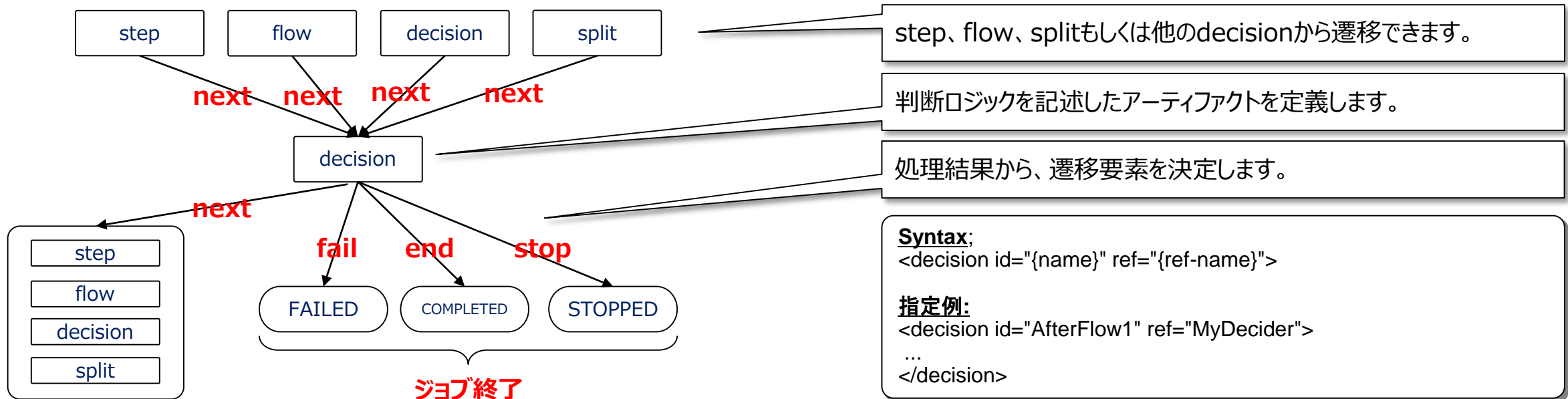
4.4. 実行要素 - split

- splitは、**同時に処理するflow**を定義するために使用されます。
- splitは、flowのみを子エレメントとして持つことができます。各子flowは異なるスレッドで実行されます。
- splitの子エレメントとして定義された全てのflowの実行が終了した段階で、splitの処理が完了します。
- splitの次は、step、flow、decision、もしくは他のsplitに遷移できます。



4.4. 実行要素 - decision

- decisionを用いることで、順次処理で**次に行うべき処理の判断をカスタマイズ**することが可能です。
- decisionは、step、flow、splitもしくは他のdecisionのnextエレメントのターゲットとして指定します。
- decisionとして、判断のロジックをもつアーティファクトを指定する必要があります。
- decisionでは、判断の根拠にstop、fail、end、next エレメントを使うことができます。
 - 戻り値は、現在のジョブのexit statusにもセットされることになります。
- decisionの例外処理
 - decider（decisionに登録されたアーティファクト）で例外が発生すると、バッチはFAILEDのステータスで終了します。



4.5. ジョブおよびステップのステータス (1/3) - バッチ・ステータス、Exitステータスとは？

■ バッチ・ステータス と Exitステータス

- バッチジョブの実行ステータスは、ジョブ単位、およびステップ単位で管理されます。
- これらのステータスには、ランタイムが管理する「**バッチ・ステータス**」と、ユーザーが定義する「**Exitステータス**」の2種類があります。
- **バッチ・ステータス**はランタイムによって決定されますが、**Exitステータス**はJSLやプログラムの中で決定されます。
 - ステップの Exitステータスはデフォルトでnullであり、StepContextオブジェクトのexit-status Setterメソッドによって値が設定されます。
 - 特にバッチ・アーティファクトによって明示的に設定されなかった場合は、ランタイムによってデフォルトのステータスが設定されます。
 - バッチ・ステータスの一覧は次ページを参照ください。

■ ジョブの停止条件とステータスの関係

- ステータス：**COMPLETED**
 - ジョブ・レベルの実行エレメント（step、flow、split）が、next指定なし、もしくはtransitionエレメントでの指定ステータス以外で終了した場合、ジョブは**COMPLETED**に設定されます。
- ステータス：**FAILED**
 - stepがランタイムに、skipもしくはretry対象とならない例外をスローした場合、ジョブは**FAILED**に設定されます。
 - また、パーティション化された、もしくは同時実行（split）されたステップが処理中の場合、それら実行中のインスタンは継続して処理を行いますが、最終的には**FAILED**で終了します。
- ステータス：**STOPPED、COMPLETED、FAILED**
 - step、flow、decisionが**stop**、**end**、**fail**エレメントで終了した場合、ジョブは、それぞれ**STOPPED、COMPLETED、FAILED**で終了します。

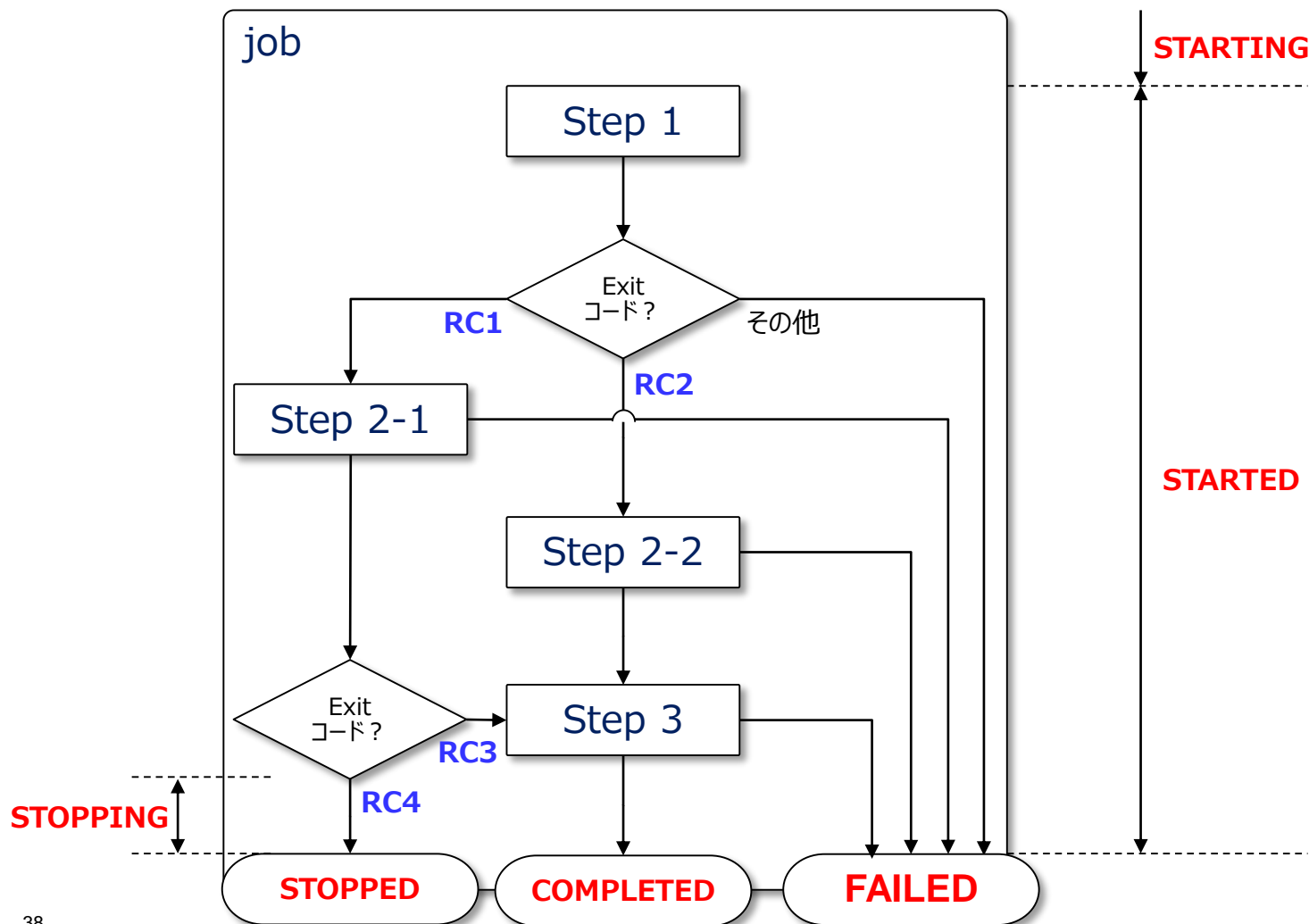
4.5. ジョブおよびステップのステータス(2/3) - バッチ・ステータス一覧

- 以下はバッチ・ステータスの一覧です。

ステータス	説明
STARTING	JobOperatorのスタートおよびリスタートの操作によって、バッチジョブがランタイムに渡された状態です。実際にジョブが実行される前にこの状態になります。
STARTED	ランタイムによってバッチジョブの処理が開始された状態です。処理が開始されるとこの状態になります。
STOPPING	JobOperatorのストップもしくは、JSLの<stop>エレメントによって、ジョブの停止が要求されている状態です。JobOperatorが停止要求を受けるとすぐにこの状態になります。
STOPPED	JobOperatorのストップもしくは、JSLの<stop>エレメントによる停止要求によって、ジョブが停止した状態です。ランタイムによってジョブが停止された後にこの状態になります。
FAILED	バッチジョブが解決されない例外もしくはJSLの<fail>エレメントによって、停止された場合にこの状態になります。
COMPLETED	バッチジョブが正常終了、もしくは<end>エレメントによって停止した場合にこの状態になります。
ABANDONED	JobOperatorのabandon操作によってマークされるとこの状態になります。abandon（破棄）されたジョブは、処理も再処理もできませんが、履歴としては残ることになります。

4.5. ジョブおよびステップのステータス(3/3) - ステータス遷移例・イメージ

- 以下は、ジョブおよびステップのステータス遷移の模式的にあらわした図です。（厳密なタイミングについては前頁の説明を参照ください。）
 - ここでは、**赤字でジョブのバッチ・ステータス**、**青字でステップのExitステータス**を表しています。Exitステータスはステップのフロー制御で使用されています。



指定例:

```

<step id="step1" >
  <fail on="FAILED" />
  <next on="RC1" to="Step2-1" />
  <next on="RC2" to="Step2-2" />
</step>
  
```

```

<step id="step2-1" >
  <fail on="FAILED" />
  <next on="RC3" to="Step3" />
  <stop on="RC4" />
</step>
  
```

```

<step id="step2-2" >
  <fail on="FAILED" />
  <next on="*" to="Step3" />
</step>
  
```

```

<step id="step3" >
  <fail on="FAILED" />
</step>
  
```

5. ジョブの管理・実装

ここからは、ジョブを実行する環境、ジョブの起動について解説すると共に、どのようにジョブを作成するか、について解説します。

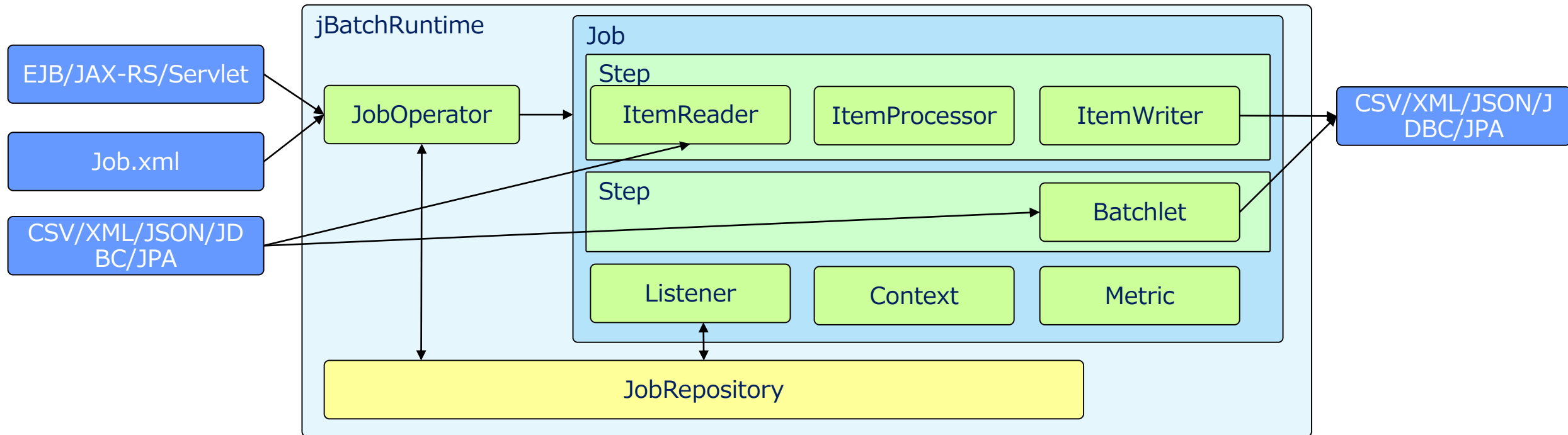
5.1. JSR-352におけるバッチ処理のランタイム

5.2. ジョブの管理

5.3. ジョブの実装

5.1. JSR-352におけるバッチ処理のランタイム

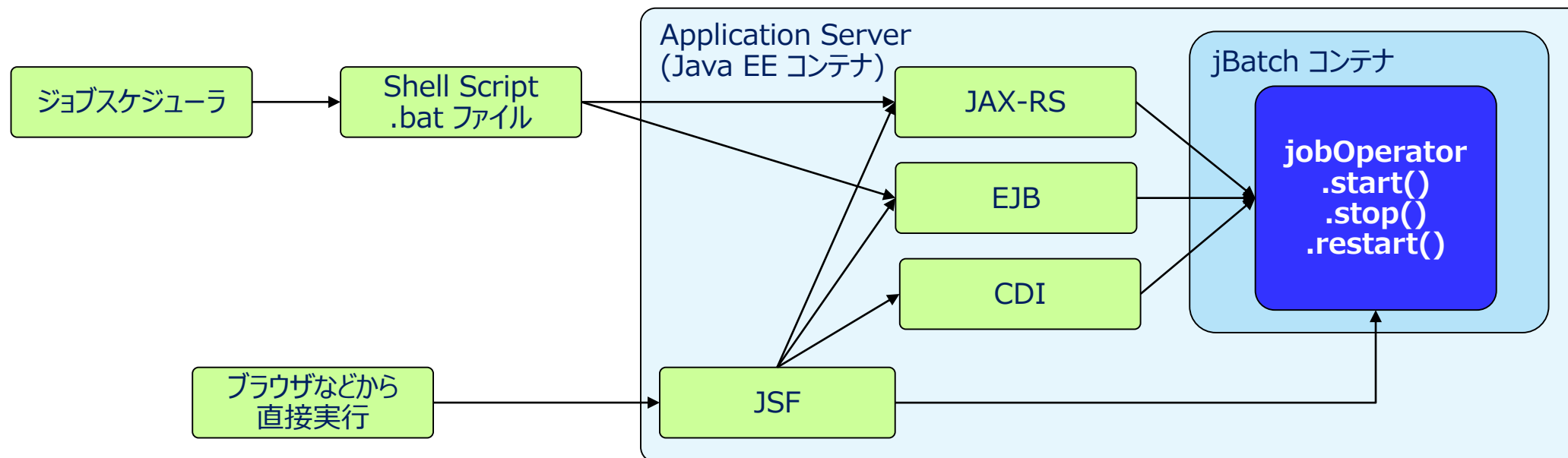
- JSR-352のバッチランタイムは、ジョブXML による処理の順序制御、チェックポイントによるレコード単位のコミット、スキップ処理などの実行環境を提供しています。ジョブ、ステップの単位でプロパティをサポートしています。
- バッチ・システムを作るためのAPI の提供に主眼を置いているため、JAX-RS やJSF などからJobOperator を呼び出してジョブを実行します。
- JSR-352のバッチランタイムが提供するプロパティ名の接頭辞にはjavax.batch が規定されています。



5.2. ジョブの管理 - ジョブオペレーター

- ジョブの開始や停止などの全体的な制御には、**ジョブオペレーター**を利用します。
 - インターフェース : `javax.batch.operation.JobOperator`
 - `JobOperator` インタフェースは、jobの開始・停止・リスタート・検査するための操作セットです。
- JSR-352には、ジョブオペレーターの呼び出し方に関する規定はありません。そのため、ジョブオペレーターを呼び出すコードが含まれるクラスを作成し、それをJAX-RSやJSF、EJBなどで実行します。また、jBatch の仕様にはバッチを決まった時間に起動する方式や、それを実行するAPI も含まれていません。cronやタスクスケジューラーなどのジョブスケジューラーと連携させて使用します。

ジョブオペレーターの呼び出しの例



5.2. ジョブの管理 - ジョブの開始

- ジョブの実行はジョブオペレーターが提供するメソッドを利用します。JobOperator には、start()、stop()、restart() などのメソッドが含まれています。JobOperator インターフェースは、BatchRuntime クラスの getJobOperator() メソッド経由でコンテナから取得します。
- ジョブの開始は、ジョブXML を指定して起動します。start() の第一引数はJobId(Job.xml 名)を、第二引数にはジョブのプロパティを指定します。ここで指定したプロパティは、Job.xml の中やジョブを構成するクラスの中で取得可能です。返り値はジョブを実行すると割り振られるJobExecution のId です。

ジョブ実行の例

```
Properties p = new Properties();  
...  
JobOperator jobOperator = BatchRuntime.getJobOperator();  
//start  
long jobExecId = jobOperator.start("JobSample" , p);
```

Job.xml 名とジョブプロパティを引数にして起動します。ジョブの実行は BatchRuntimeを介して行います。

Job.xml の配置については、「[5.3. ジョブの実装 - Job XMLのロード](#)」を参照ください。

JobSample.xml

```
<?xml version="1.0" encoding="UTF-8"?>  
<job id="JobSample"  
....  
</job>
```

5.2. ジョブの管理 - ジョブランタイムId

- 最初にジョブを実行するとコンテナ内部で**JobInstance** と**JobExecution** が生成されます。その際に、それぞれにId が割り振られます。JobInstance はジョブの1回分を、JobExecution は実行そのものを表しています。
- 以下のId でジョブランタイムを定義します。instanceId, executionId, stepExecutionId は、ジョブリポジトリ内で一意の値である必要があります。

Id	説明
instanceId	ジョブのインスタンスを表すIdです。新しいジョブインスタンスは、バッチランタイムからJobOperator を取得し、start メソッドを呼び出して開始することができます。
executionId	特定のジョブインスタンスを実行する際に使用するId です。ジョブのリスタート、停止時に引数に指定します。JobOperator を取得してrestart、stop メソッドを呼び出します。
stepExecutionId	ジョブの中で特定のステップを実行するIdです。

ジョブの実行例（停止・再起動）

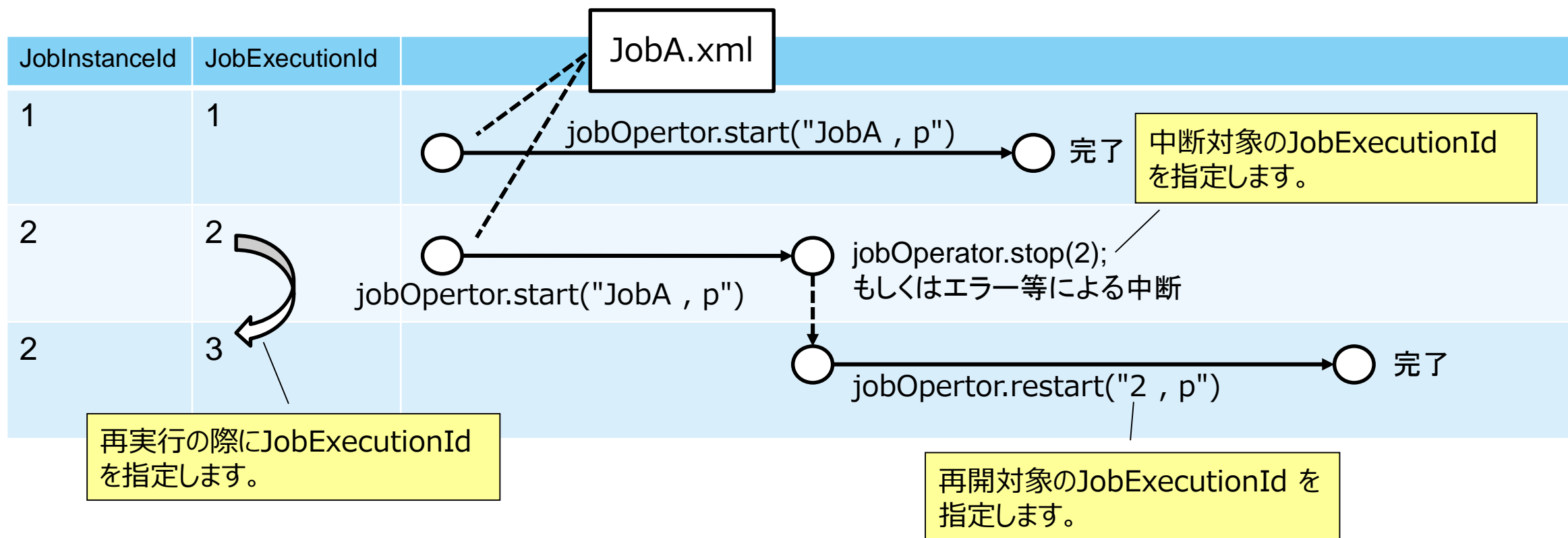
```
...
//stop
long jobExecId = jobOperator.stop( execId );
//restart
long jobExecId = jobOperator.restart( execId , p);
```

executionId を指定してジョブを停止します。executionId はstart 実行時の返り値です。

executionId とジョブプロパティを引数にして再起動します。

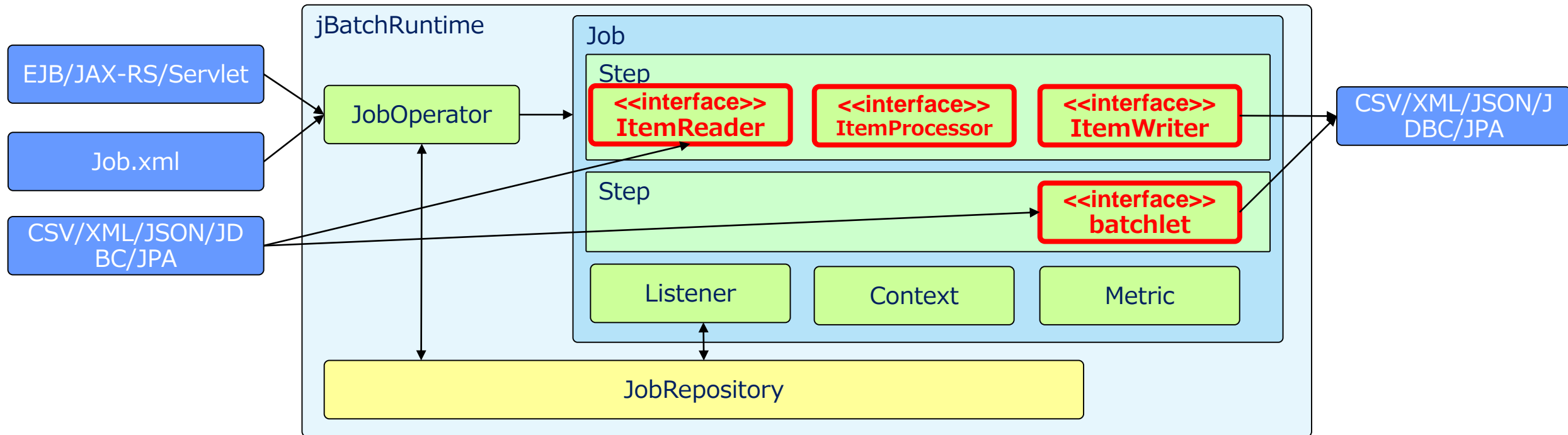
5.2. ジョブの管理 - JobInstance とJobExecution の関係

- ジョブがエラーにより中断したり、stop() により明示的に中断が指示された場合は、そのジョブに対してrestart() を指示することができます。この場合、再開前のJobInstance に対して、別のJobExecution のインスタンスが生成されます。そのJobExecution には新たなJobExecutionId が生成されます。こうした各ジョブの状態や履歴に関する情報はジョブリポジトリに保存されます。JobOperator のstop() やrestart() を実行するときの引数には、**JobExecutionId** を指定します。



5.3. ジョブの実装 - jBatchのランタイム インターフェース

- 次頁から以下4つのインターフェースについて記載します。
 - chunk方式ステップ
 - ItemReader、ItemProcessor、ItemWriter
 - batchlet方式ステップ
 - batchlet



5.3. ジョブの実装 - chunk方式ステップの実装 ItemReader

■ ItemReader

- インターフェース : `javax.batch.api.chunk.ItemReader`
- ItemReader は chunk 型ステップで加工や計算を行うための情報をデータベースやファイルなどから読み込みます。

■ 以下のメソッドが規定されています。

- `open()`
 - 情報が格納されているリソースを開く処理を実装するためのメソッドです。
 - 引数にはチェックポイントの情報を指定します。
- `close()`
 - `open()` でアクセスしたリソースを閉じる処理を実装するためのメソッドです。
- `readItem()`
 - `open()` で開いたリソースからレコードを1件ずつ読み取る処理を実装するためのメソッドです。引数はなく、戻り値は `Object` 型です。この戻り値がそのまま `ItemProcessor` に渡されます。読み取るレコードがなくなったとき、戻り値 `null` を返し、`ItemProcessor` にレコードが渡されなくなります。
- `checkpointInfo()`
 - 「[5.3. ジョブの実装 - chunk方式ステップの実装 ItemWriter](#)」参照

5.3. ジョブの実装 - chunk方式ステップの実装 ItemProcessor

■ ItemProcessor

- インターフェース : `javax.batch.api.chunk.ItemProcessor`
- `ItemReader` で読み込まれた情報を1件ずつ受け取り、加工や計算などの処理をする役割を担います。
- 以下のメソッドのみが規定されています。
 - `processItem()`
 - 引数には、`ItemReader` の`readItem()` で戻り値として設定された情報が格納されます。
 - `processItem()` の戻り値は`ItemWriter` の`writeItem()` に渡す情報です。
 - チェックポイントが実行されるまで`processItem()` の戻り値はアプリケーションサーバーの中で保持されています。仮に前回のチェックポイントから数件を処理した後、次のチェックポイントをむかえる前にハンドリング不可能なエラーが発生した場合、トランザクションはロールバックされて内部的に保持していた情報はクリアされます。
 - 再実行した場合は、前回のチェックポイント時点（`ItemReader` の`checkPointInfo()` で記録された時点）から再開されるため、クリアされても問題はありません。

5.3. ジョブの実装 - chunk方式ステップの実装 ItemWriter

■ ItemWriter

- インターフェース : `javax.batch.api.chunk.ItemWriter`
- `ItemProcessor`で処理されたデータを書き出す処理を行います。

■ 以下のメソッドが規定されています。

- `open()` : `ItemReader` と同様。
- `close()` : `ItemReader` と同様。
- `writeItem()`
 - `ItemProcessor` の `processItem()` で処理した結果を受け取り、データベースやファイルに書き出す処理を実装します。
 - チェックポイントのタイミングで呼び出されるため、複数件のアイテムが `Object` 型、`List` 形式で渡されます。
- `checkpointInfo()`
 - 処理の途中経過を記録するためのメソッドです。`checkpointInfo()` の戻り値は、コンテナが内部で保持しているデータストアに永続化されます。そのため、戻り値は `Serializable` インターフェースを実装したクラスである必要があります。
 - 永続化された情報は `open()` の引数として渡されます。初回起動時、もしくはチェックポイントの前に終了したバッチの再実行時、あるいは `checkpointInfo()` の戻り値として `null` を返していた場合は、この値は `Null` になります。`checkpointInfo()` で `Null` 以外の値を返していた場合は再実行時の `open()` の引数にその値がセットされます。

5.3. ジョブの実装 - batchlet方式ステップの実装

■ Batchlet

– インターフェース : `javax.batch.api.Batchlet`

■ 以下のメソッドが規定されています。

– `process()`

- `batchlet`が担う機能を実装するためのメソッドです。引数はなく、戻り値はString 型です。
- 戻り値に指定した文字列は、このステップの終了ステータスとしてコンテナに解釈されます。

– `stop()`

- `batchlet`の処理中に(`process()` の実行中)に、ジョブ全体の中断が指示された場合に呼び出されるメソッドです。
- このメソッドは、`process()` を実行しているスレッドとは別のスレッドで実行されているため、`process()` の処理に直接割り込むことはできません。
- 注意点として、ジョブ全体の中断が指示されても、`process()` の処理が中断されない場合もあります。JobOperator クラスの`stop()` メソッドにより指示します。

5.3. ジョブの実装 - ジョブメトリック

- **メトリック**はchunk型のステップに対して実行時の統計情報を提供します。
 - インターフェース : `javax.batch.runtime.Metric`
- 提供される情報は読み取りレコード数、書き込みレコード数、スキップが発生した数など、処理量に関するものです。
- 以下のようなメトリックを提供しています。これらのメトリックは`StepExecution` クラスから呼び出します。
 - インターフェース : `javax.batch.runtime.StepExecution`

メトリック	説明
<code>readCount</code>	読み取りが成功したレコード数
<code>writeCount</code>	書き込みが成功したレコード数
<code>filterCount</code>	データの加工などの処理をしたレコード数
<code>commitCount</code>	トランザクションのコミットしたレコード数
<code>rollbackCount</code>	トランザクションのロールバックしたレコード数
<code>readSkipCount</code>	読み取り時にスキップが発生したレコード数
<code>processSkipCount</code>	データの加工などの処理時にスキップが発生したレコード数
<code>writeSkipCount</code>	書き込み時にスキップが発生したレコード数

5.3. ジョブの実装 - JobContext

■ JobContext

– インターフェース : `javax.batch.runtime.context.JobContext`

- 現在のジョブ全体に関する情報を提供する役割を担います。必要に応じて任意の値（`UserData`）をセットすることもできます。`JobContext` のインスタンスは右記のように定義することでコンテナによってインジェクトされます。

```
@Inject  
private JobContext jctx ;
```

- 以下は`JobContext` インターフェースが持つメソッドです。詳細は`JavaDoc` を参照してください。

1. ジョブ全体の名前やID に関する情報の取得

- `getJobName()`、`getExecutionId()`、`getInstanceId()`

2. ジョブ全体のステータスに関する情報の取得と登録

- `getBatchStatus()`、`getExitStatus()`、`setExitStatus()`

3. ジョブ全体の属性に関する情報の取得と登録

- `getProperties()`、`getTransientUserData()`、`setTransientUserData()`

– `JobContext` と `StepContext` はステップを実装するクラスのコンストラクタの中からはアクセスすることはできません。各クラスのコンストラクタは、コンテナによりインスタンスが生成される時点で実行されますが、その時点ではまだコンテキストは生成されていません。そのため、ジョブ全体の初期処理はコンストラクタではなく、`open()` の中で実装します。

(参考) Java Documentation - Interface `JobContext`

- <https://docs.oracle.com/javaee/7/api/javax/batch/runtime/context/JobContext.html>

5.3. ジョブの実装 - StepContext

■ StepContext

- インターフェース : `javax.batch.runtime.context.StepContext`
- 現在のステップ全体に関する情報の提供および設定を行います。
- 以下はStepContext インターフェースが持つメソッドです。詳細はJavaDoc を参照してください。太字はJobContextに同等のメソッドがないものです。
 1. ステップ全体の名前やID に関する情報の取得
 - `getStepName()`、`getStepExecutionId()`
 2. ステップ全体のステータスに関する情報の取得と登録
 - `getBatchStatus()`、`getExitStatus()`、`setExitStatus()`、**`getExecution()`**、**`getMetrics()`**
 3. ステップ全体の属性に関する情報の取得と登録
 - `getProperties()`、`getTransientUserData()`、`setTransientUserData()`、**`getPersistentUserData()`**、**`setPersistentUserData()`**
- `getExecution()` : ステップ内で最後に発生した例外クラスが取得できるメソッドです。`getPersistentUserData()`、`setPersistentUserData()` : ステップ内部で共通に利用する任意の値の設定と取得ができます。set メソッドを通して設定された値は、コミットのタイミングで永続化されます。一度永続化されたものはジョブが一度停止し、再実行した際でもget メソッドを通して取得することができます。
 - (参考) Java Documentation - Interface StepContext
 - <http://docs.oracle.com/javaee/7/api/javax/batch/runtime/context/StepContext.html>

5.3. ジョブの実装 - StepListener とJobRepository

■ StepListener

- インターフェース : `javax.batch.api.listener.StepListener`
- ステップの開始直前と終了直後のタイミングで、任意のコードを実行させることができます。
- 以下のメソッドが規定されています。
 - `beforeStep()`
 - リスナが設置されたステップが実行される直前に実行されるメソッドです。
 - `afterStep()`
 - リスナが設置されたステップが実行される直後に実行されるメソッドです。

■ JobRepository

- 実行中のジョブの情報や、実行が終わったジョブに関する情報は、ジョブリポジトリに保存されます。jBatch の仕様では、保存先や保存期間などに関する規定は特にありません。一般的にはコンテナ内部で保持しているデータベースなどに永続化され、アプリケーションサーバーを停止・再起動したあとでも過去の情報を取得できるようにします。
- jBatch のAPI には、ジョブリポジトリそのものを表現するようなクラスやインターフェースは用意されていません。情報の更新はコンテナによって自動的行われ、情報の取得はジョブオペレーター、コンテキスト、メトリックのAPI を利用します。

5.3. ジョブの実装 - バッチ・ステータスと終了ステータス

- JSR-352では処理結果を表現するものとして、バッチ・ステータスと終了ステータスを提供しています。
 - 概要については「[4.5. ジョブおよびステップのステータス \(1/3\) - バッチ・ステータス、Exitステータスとは？](#)」に説明がありますが、ここでは設定のためのメソッド等について説明します。
- **バッチ・ステータス**
 - バッチ・ステータスは、コンテナにより自動的に設定される文字列です。設定される値は、JSR-352の仕様としてあらかじめ定義されています。
 - バッチ・ステータスの値はgetBatchStatus() により取得できます。このメソッドはJobContext、JobExecution、StepContext、StepExecution の各クラスに存在します。いずれのクラスにもsetter メソッドは存在しません。戻り値はEnum 型で、javax.batch.runtime.BatchStatus として定義されています。
 - バッチ・ステータスの値は、**STARTING、STARTED、STOPPING、STOPPED、FAILED、COMPLETED、ABANDONED** があります。（それぞれの説明については、「[4.5. ジョブおよびステップのステータス\(2/3\) - バッチ・ステータス一覧](#)」参照）
- **Exitステータス**
 - 処理結果を詳細に設定することが可能な文字列です。**設定しない場合はバッチ・ステータスと同じ文字列がコンテナにより自動的に設定されます。**
 - バッチに対する終了ステータスの設定は、JobContext のsetExitStatus() を用います。ステップに対する終了ステータスの設定は、StepContext のsetExitStatus() を使います。

5.3. ジョブの実装 - バッチクラスのロード

- JSR-352では以下 3 つのローダーを規定しています。記述順にロードが試行されます。

(実際には、WAS Libertyなどのランタイムの実装に依存します。)

1. 実装固有のローダー(implementation-specific loader)

- バッチランタイムの実装は、ジョブXML のバッチによる参照クラスの解決とインスタンス化の手段を実装固有の方法とすることが可能です。バッチランタイムがバッチクラスをインスタンス化しようとする時、実装固有のメカニズム（が一つでも存在すれば）が最初に試行されます。ローダーはインスタンスもしくはnull を必ず返します。
実装固有ローダーはCDI やSpring DI などが挙げられます。

2. アーカイブローダー(archive loader)

- 実装固有のメカニズムが存在しないかバッチクラス参照の解決に失敗する（nullを返す）場合、バッチランタイムの実装はアーカイブローダーで参照を解決します。ランタイムの実装は、参照名を実装クラス名にマッピングするbatch.xml ファイルを基にして、ルックアップを解決するアーカイブローダーを提供しています。

3. スレッド・コンテキスト・クラスローダー(thread context class loader)

- アーカイブローダーがバッチクラス参照の解決に失敗する（nullを返す）場合、バッチランタイム実装はスレッドコンテキストクラスローダーを通してクラスの参照を解決してロードします。

5.3. ジョブの実装 - Job XMLのロード

- Job XMLは、ジョブ開始時のJobOperator.startメソッドの引数として渡します。
- 指定されたJob XMLファイルは、以下のローダーの順序によって読み込まれます。

1. 実装固有ローダー(implementation-specific loader)

- バッチランタイムの実装はJob XML による参照をJob XML ドキュメントに実装固有の手段で提供することが可能です。
- 実装固有ローダーの目的は、リポジトリ・ファイルシステム・リモートキャッシュなどの、アプリケーションアーカイブの外側からXML ロードを可能にするためです。

2. アーカイブ・ローダー(archive loader)

- 実装固有メカニズムがJob XML による参照の解決に失敗した場合、バッチランタイム実装はアーカイブローダーで参照を解決します。実装はMETA-INF/batch-jobs ディレクトリからルックアップして参照を解決するアーカイブローダーを提供します。
- Job XML ドキュメントは開発者がMETA-INF/batch-jobs ディレクトリ下にアプリケーション(WEB-INF/classes/META-INF/batch-jobsの.warファイル)と共にパッケージ化します。

5.3. ジョブの実装 - アプリケーション・パッケージ・モデル

- jBatch はWeb アプリケーションと同様に、構成するファイル群をwar ファイル形式などにまとめてコンテナにデプロイします。ファイル形式によってジョブXML は以下のディレクトリーに配置してパッケージする必要があります。
 - jar の場合 : META-INF/batch-jobs/
 - war の場合 : WEB-INF/classes/META-INF/batch-jobs/
- ジョブXML のファイル名とジョブID は一致させる必要があります。

例 : **JobA.xml**のフォーマット

```
<?xml version="1.0" encoding="UTF-8"?>
<job id="JobA" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
    ....
</job>
```

ジョブXMLのファイル名とジョブIdを一致させます。

5.3. ジョブの実装 - クラス一覧

- 以下はJSR-352ランタイムで提供されているクラスです。
 - JobContext
 - StepContext
 - Metric
 - PartitionPlan
 - BatchRuntime
 - BatchStatus
 - JobOperator
 - JobInstance
 - JobExecution
 - StepExecution
 - Batch Exception Classes

- 詳細は以下を参照してください。
 - (参照)JSR_352-v1.0_Rev_a-Maintenance_Release.pdf – 10.9 Supporting Classes
 - http://download.oracle.com/otn-pub/jcp/batch-1_0_revA-mrel-spec/JSR_352-v1.0_Rev_a-Maintenance_Release.pdf?AuthParam=1451183891_d8aab26344cecb4fff10993b245f74ab

6. サンプル・アプリケーション

それでは、ここまで解説した内容をベースに、WAS Libertyでサンプルのバッチアプリケーションを作成し、動かしてみます。

6.1. 開発環境について

6.2. 開発の流れ

6.3. 手順解説

6.1. 開発環境について

- まずは、以下の開発環境が準備されていることを確認してください。
- 前提
 - WebSphere Liberty Profile (WLP) が導入済みであること。
 - WLP のバージョンは8.5.5.6 以降であること。
 - フィーチャー : batchManagement-1.0 が導入済みであること。
 - 接続するDB を準備してあること。

6.2. 開発の流れ

- ここからはIBM WebSphere Application Server Liberty Profile (以下WLP) をランタイムとしてWebSphere Developer Tools(以下WDT)を使ったJSR-352バッチアプリケーション開発の一連の流れを紹介します。
- WDT はJSR-352 に基づいたバッチアプリケーションを作成するツールを提供しています。当資料では、このバッチプロジェクト作成のためのツールのセットアップ、バッチプロジェクトの作成、ジョブの作成、WLP へのデプロイ、ジョブの実行、をガイドします。詳細は次頁からご紹介します。
 1. Eclipse とWDT の導入
 2. Java batch tools の導入
 3. WLP の構成
 4. データベース接続の設定
 5. バッチプロジェクトの作成
 6. ジョブの作成
 7. batchletステップの作成
 8. デプロイ
 9. ジョブの実行
 10. ジョブの実行結果

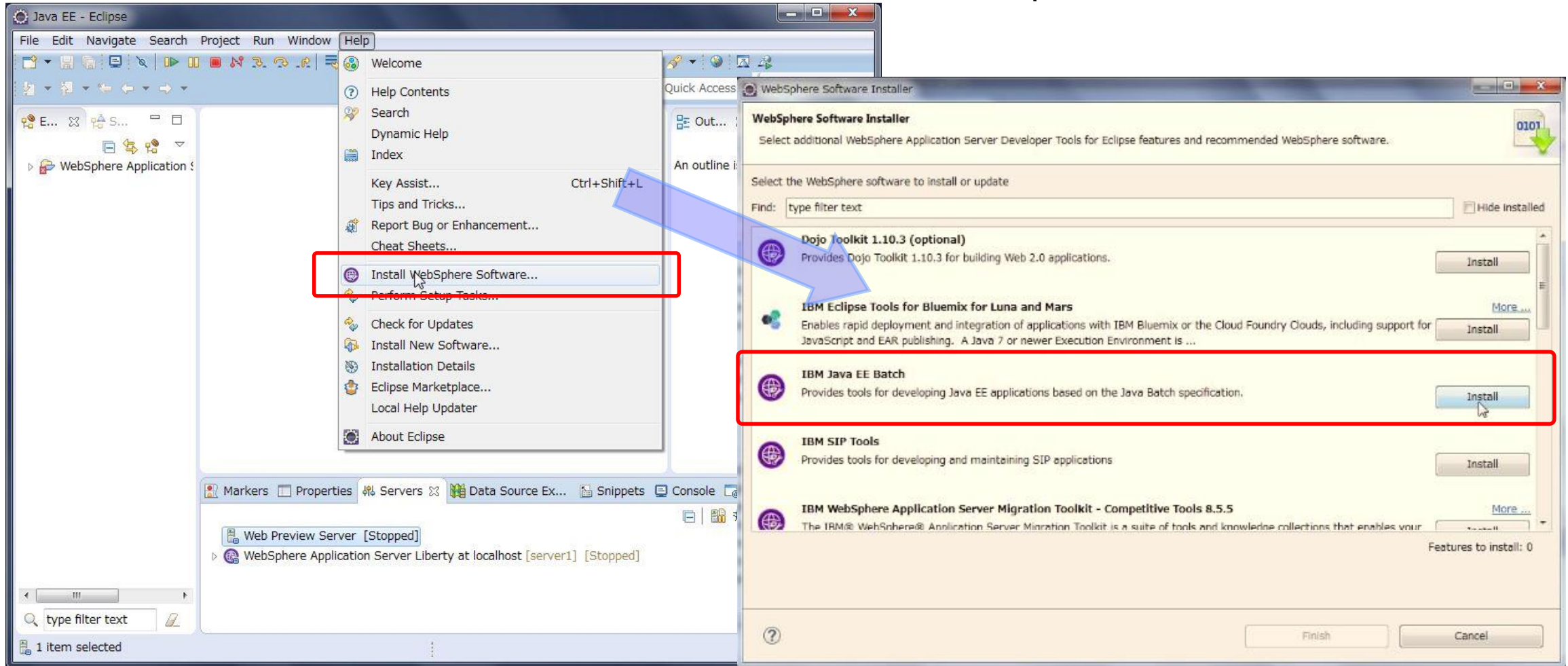
6.3. 手順解説 - 1. Eclipse とWDT の導入

- 以下のサイトの手順を参考にEclipseとWDTを導入します。

- http://www.ibm.com/developerworks/jp/websphere/library/was/liberty_intro/1.html

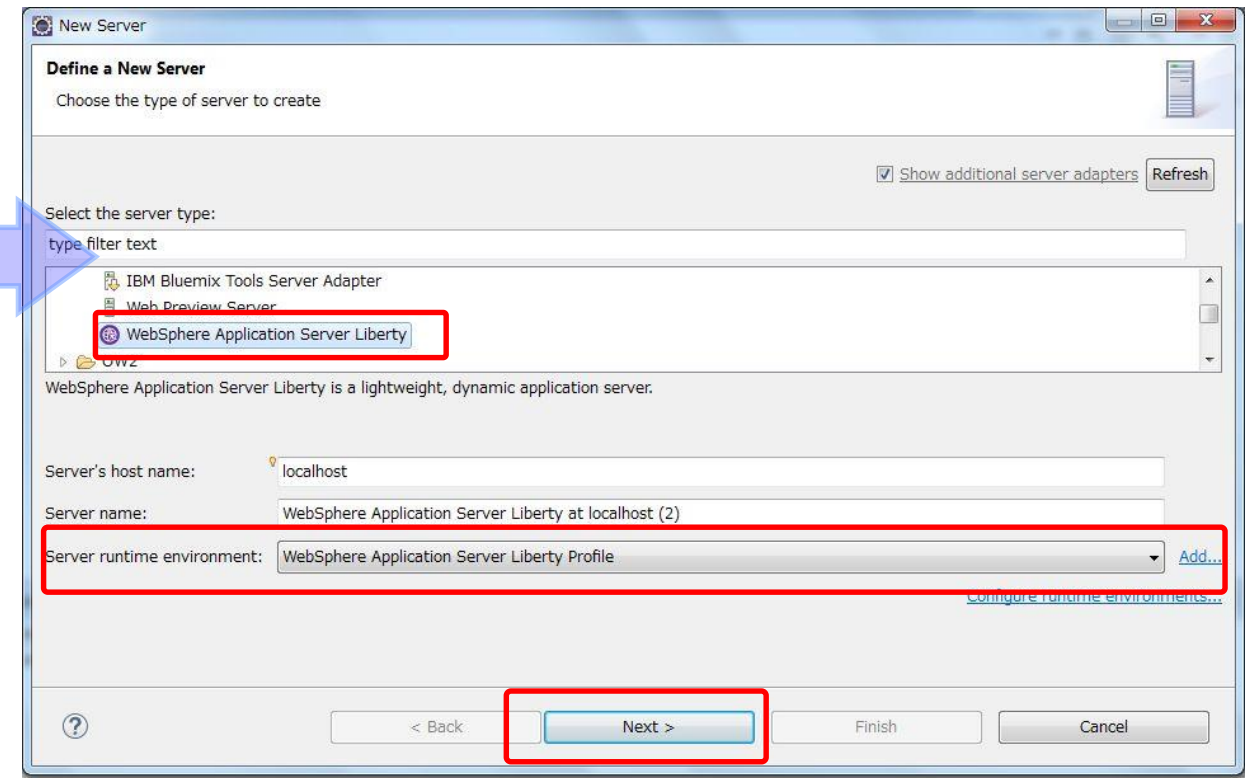
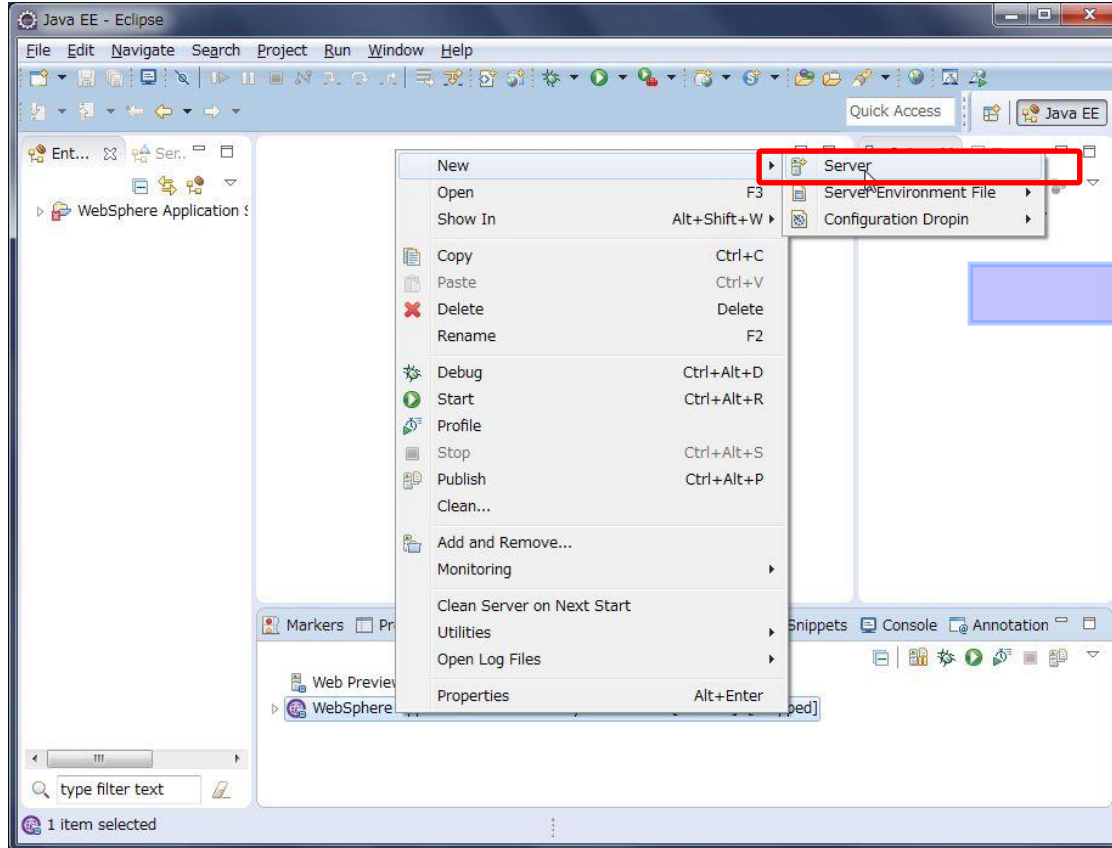
6.3. 手順解説 - 2. Java batch tools の導入

- Eclipse の「Help」 -> 「Install WebSphere Software」をクリックします。「IBM Java EE Batch」の「Install」をクリックします。ウィザードの指示に従って導入を完了させ、Eclipse を再起動します。



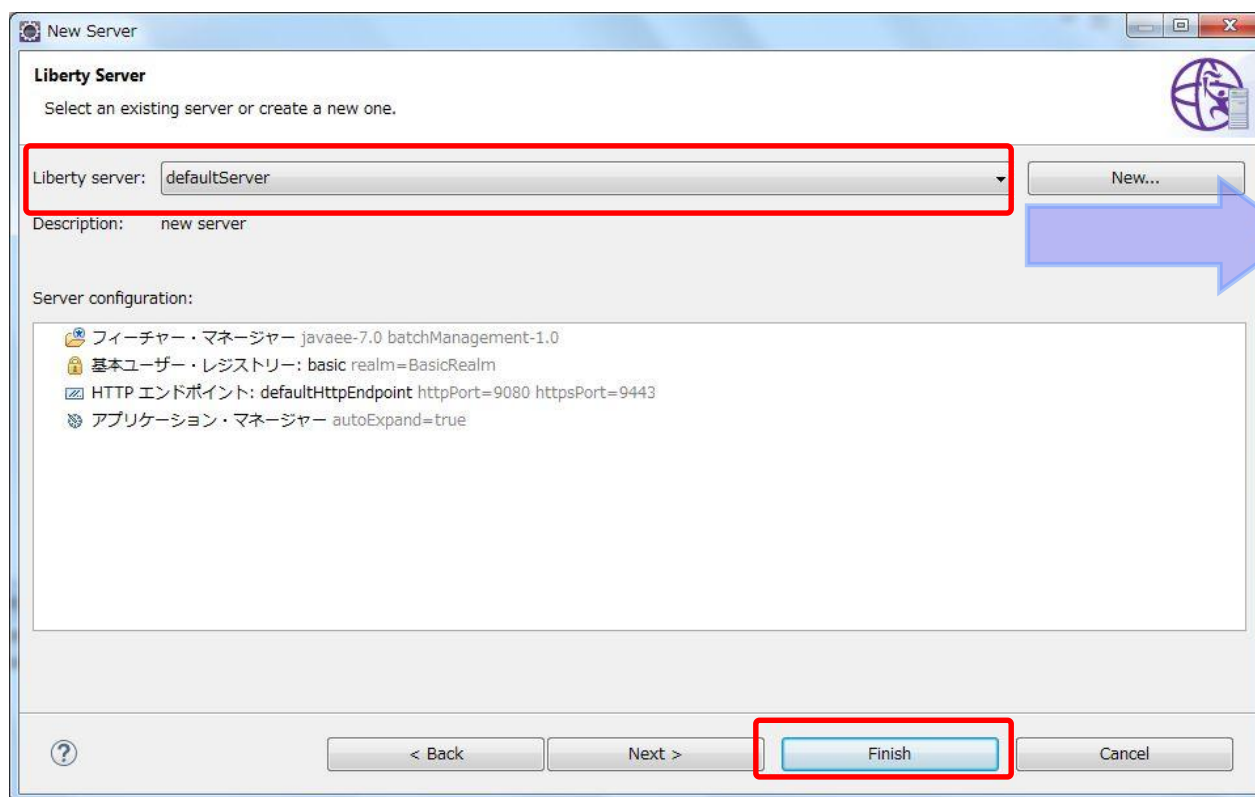
6.3. 手順解説 - 3. WLP の構成 (1)

1. Eclipse のServer ビューを右クリックして「New」->「Server」をクリックして新たにServer を作成します。
2. Select the server type から「WebSphere Application Server Liberty」を選択して、“server runtime environment”がWebSphere Application Server Liberty Profile になっていることをかくなして「Next」をクリックします。



6.3. 手順解説 - 3. WLP の構成 (2)

3. "Liberty server"を選択して、「Finish」をクリックします。
4. 作成したServer のServer.xml を開き(Server ビューのServer.xml をダブルクリックします)、batchManagement-1.0 をフィーチャーエレメントに追加します。batch-1.0 は自動的に有効になります。



server.xml

```
4      <!-- Enable features -->
5      <featureManager>
6          <feature>batchManagement-1.0</feature>
7      </featureManager>
```

5. Server ビューから作成したWLP を右クリックしてServer を開始します。

6.3. 手順解説 - 4. データベース接続の設定

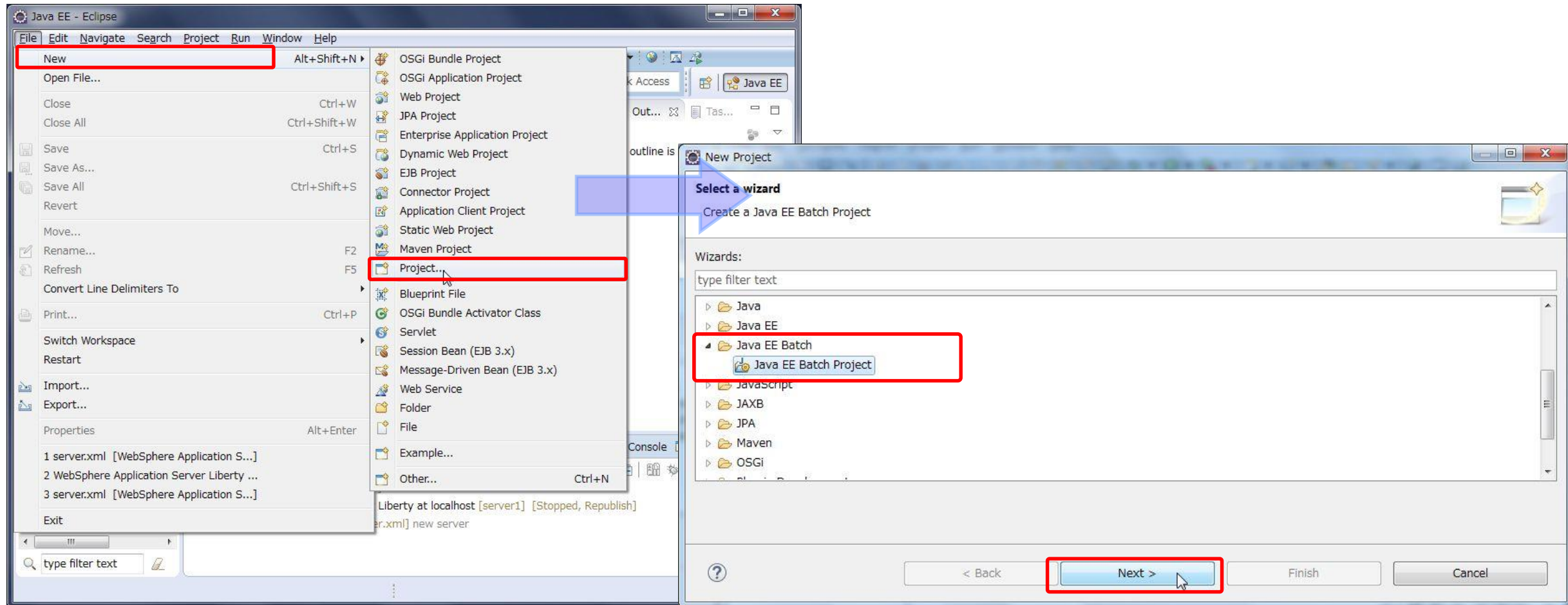
- WLP のバッチフィーチャーが適切に動作するためにはJob Repositoryとして永続ストアが必要です。バッチランタイムは必要な表を自動作成するので、server.xml に使用するDB を指定する必要があります。DB はサポートされているDBベンダーの中から選択することができます。（ここで指定したDBがバッチランタイムのJobRepository にあたります。）
1. 当資料では、Derby を使用し、createDatabase="create"に指定することで自動的にデータベースを作成します。
 2. server.xml を開いて、データソース、データストア、バッチパーシスタンスを以下のように設定します。
 - Derby database名 : \${server.config.dir}/resources/BATCHDB
 - Derby driver (derby.jar) : \${server.config.dir}/resources/derby
 - \${server.config.dir} : usr/servers/server_name (デフォルト)

server.xml

```
29 <dataSource id="batchDB">
30   <jdbcDriver>
31     <library>
32       <fileset dir="${server.config.dir}/resources/derby"; includes="derby.jar"></fileset>
33     </jdbcDriver>
34     <properties.derby.embedded createDatabase="create" databaseName="${server.config.dir}/resources/BATCHDB" password="pass" user="user"/>
35   </dataSource>
36
37   <databaseStore id="BatchDatabaseStore" dataSourceRef="batchDB" />
38
39   <batchPersistence jobStoreRef="BatchDatabaseStore" />
```

6.3. 手順解説 - 5. バッチプロジェクトの作成（１）

1. Eclipse の「File」 -> 「New」 -> 「Project」をクリックします。
2. New Project の「Java EE Batch」 -> 「Java EE Batch Project」 -> 「Next」をクリックします。



6.3. 手順解説 - 5. バッチプロジェクトの作成（2）

New Java EE Batch Project

Create a Java EE Batch Project and add it to a new or existing web project.

Project name: sleepybatchlet

Project location

☒ Use default location

Location: C:\Users\IBM_ADMIN\workspace2\sleepybatchlet

Target runtime

WebSphere Application Server Liberty Profile

Configuration

Default Configuration for WebSphere Application Server Liberty Profile

A good starting point for working with WebSphere Application Server Liberty Profile runtime. Additional facets can later be installed to add new functionality to the project.

Dynamic Web project membership

☒ Add project to a Dynamic Web project

Dynamic Web project name: SleepyBatchletSample

Working sets

☐ Add project to working sets

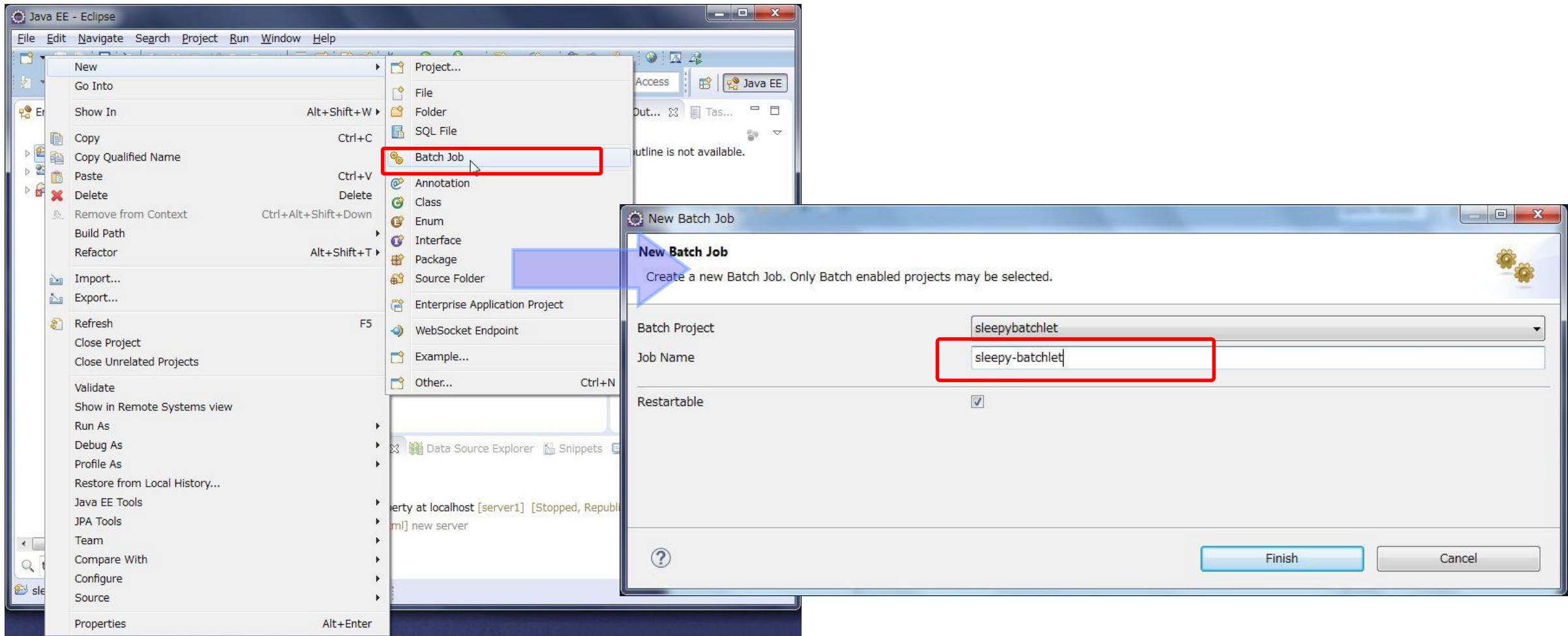
Working sets:

< Back Next > Finish Cancel

3. New Java EE Project に以下を入力して「Finish」をクリックします。
 - Project Name : sleepybatchlet
 - Target runtime : WLP
 - Dynamic web project name : SleepyBatchletSample
4. Workspace にsleepybatchlet プロジェクトが作成されます。

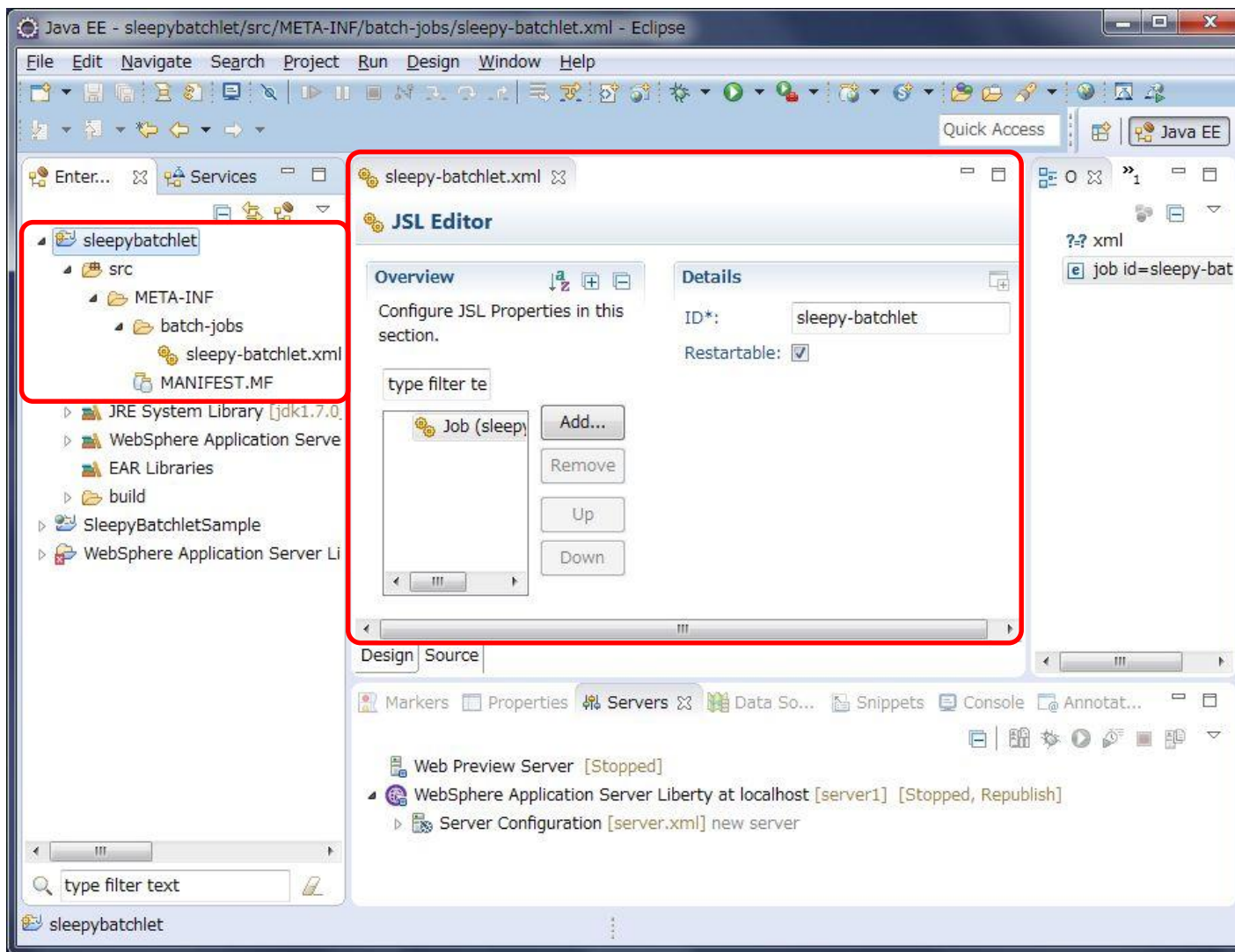
6.3. 手順解説 - 6. ジョブの作成（１）

1. sleepybatchlet プロジェクトを右クリックして「File」->「New」->「Batch Job」をクリックします。
2. Job Name に"sleepy-batchlet"を入力して「Finish」をクリックします。



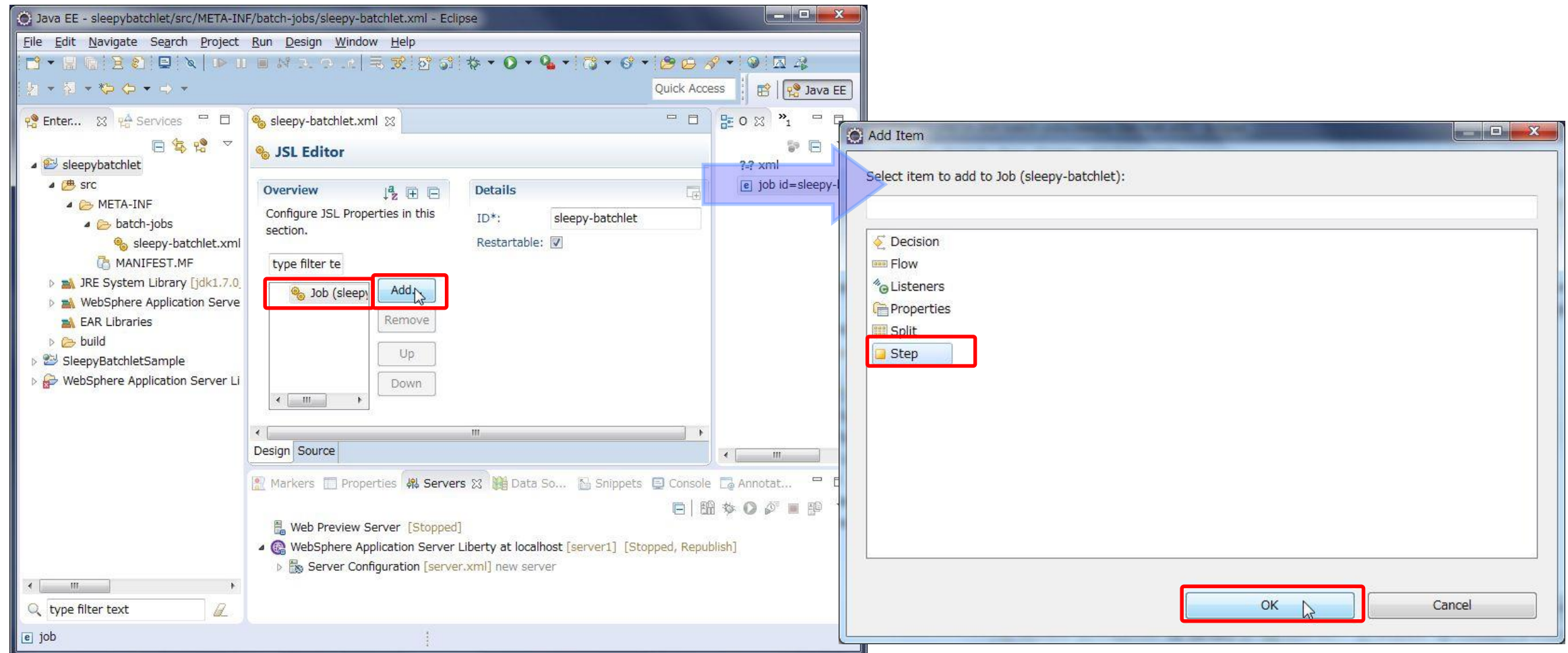
6.3. 手順解説 - 6. ジョブの作成（2）

- sleepy-batchlet.xml ファイルがsleepybatchlet プロジェクトのMETA-INF/batch-jobs フォルダに新たに作成されます。
- 自動的にJSL Editor が開きます。



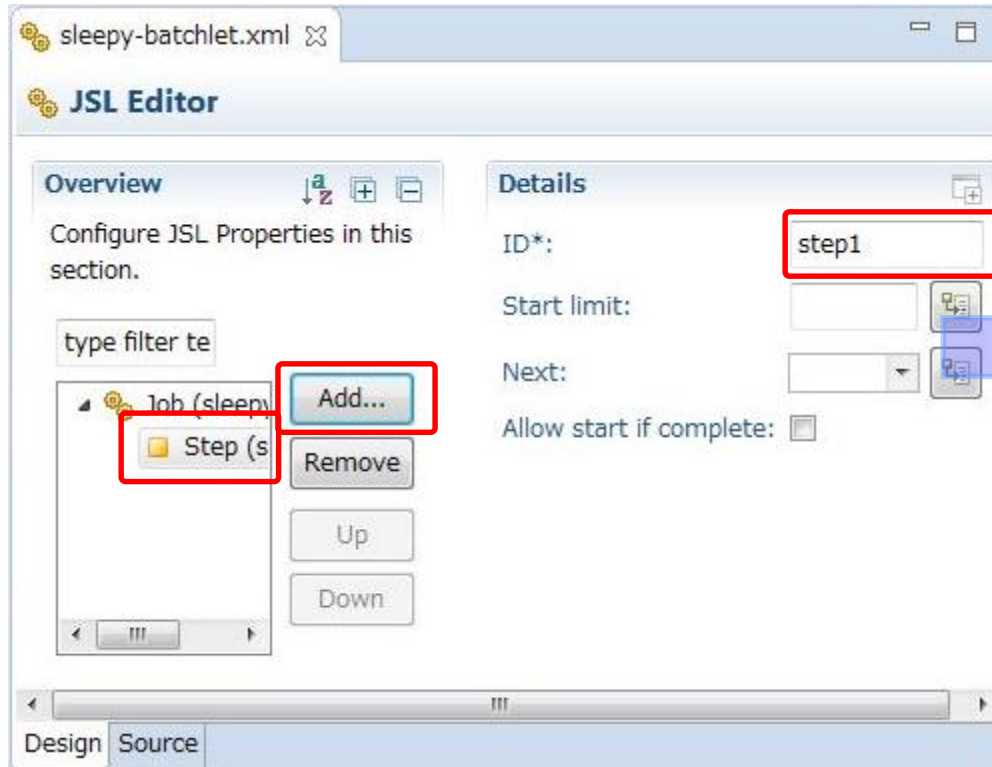
6.3. 手順解説 - 7. batchletステップの作成（１）

1. JSL Editor のJob Node を選択して、「Add」->「Step」->「OK」をクリックします。

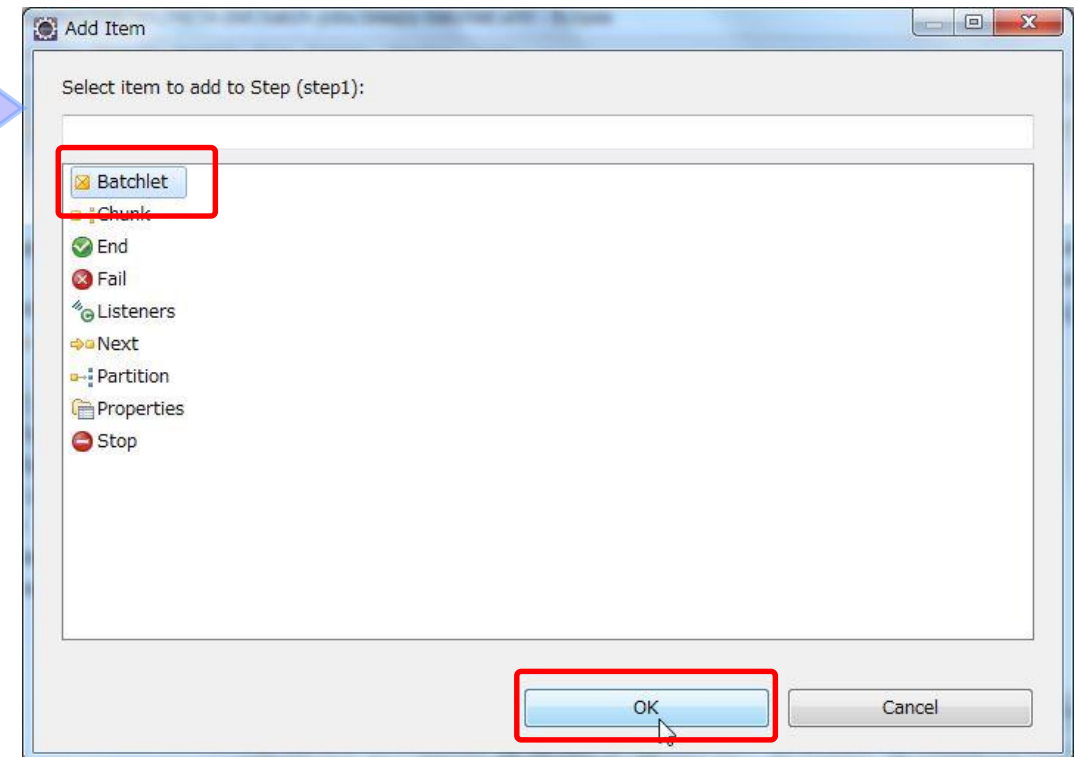


6.3. 手順解説 - 7. batchletステップの作成 (2)

2. ID を「step1」に変更します。

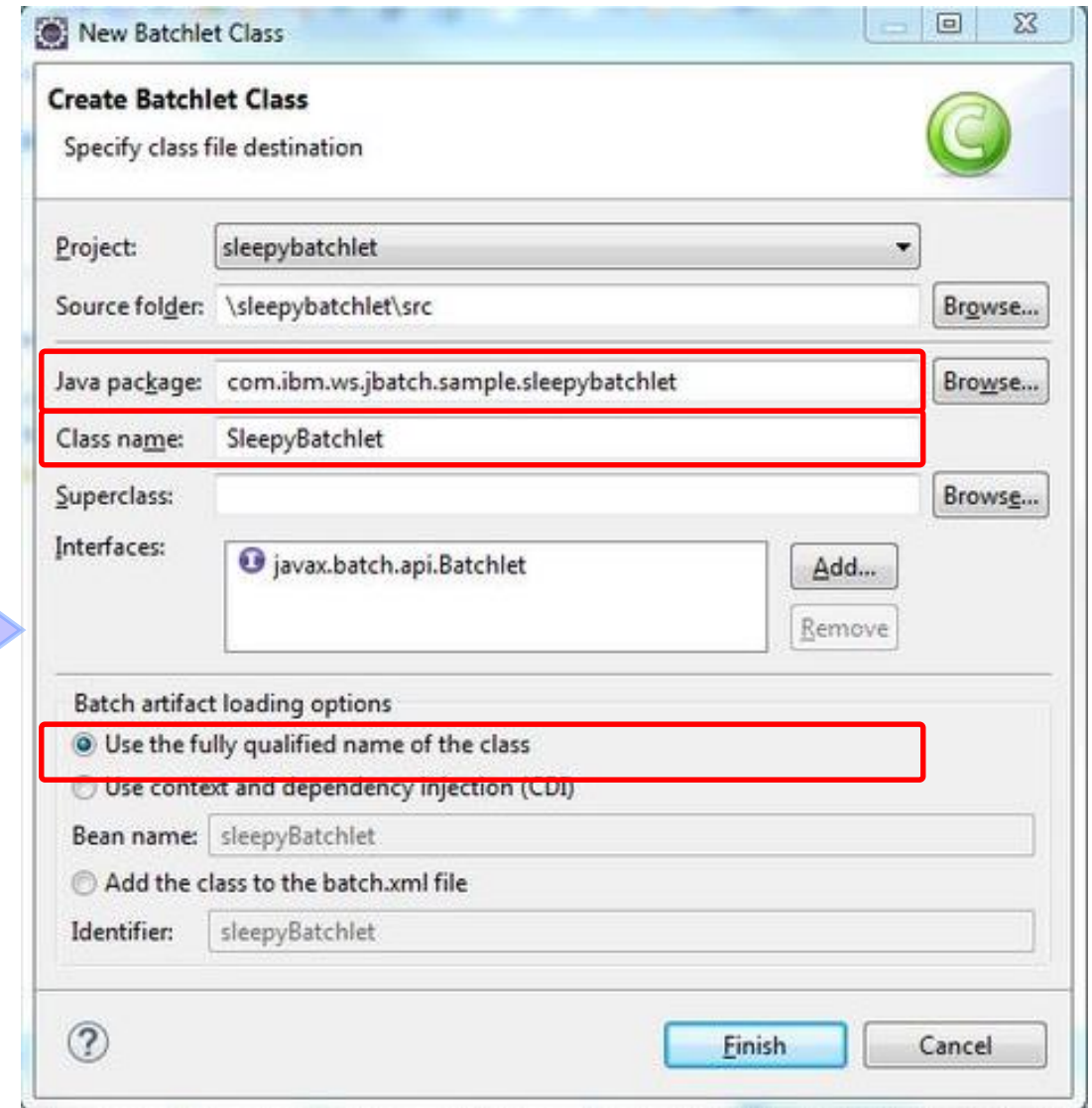
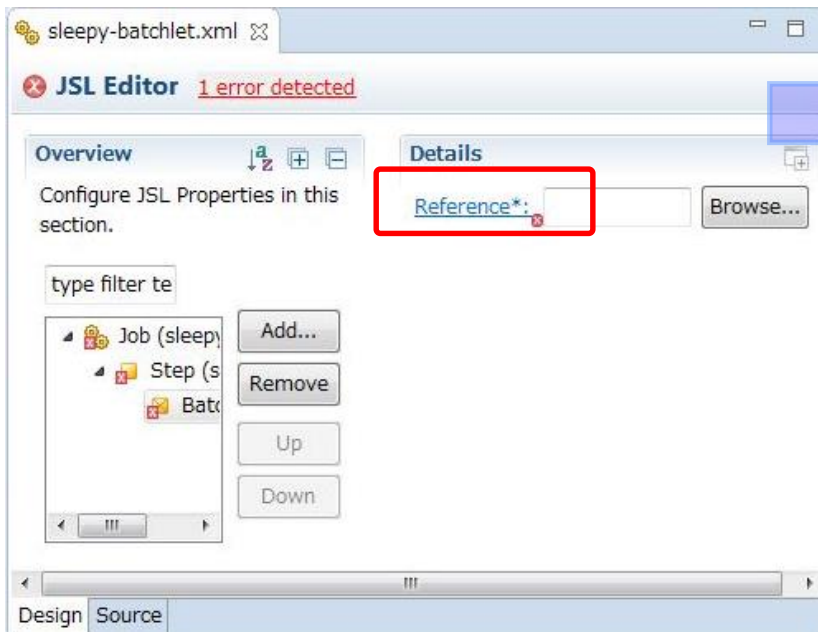


3. 「step」 -> 「Add」を選択して、「Batchlet」を選択して「OK」をクリックします。



6.3. 手順解説 - 7. batchletステップの作成 (3)

4. JSL Editor の「Reference」をクリックします。
5. New Batchlet Class に以下を入力して「Finish」をクリックします。
 - Java package :
com.ibm.ws.jbatch.sample.sleepybatchlet
 - Class name : SleepyBatchlet
 - Batch artifact loading option : デフォルト



6.3. 手順解説 - 7. batchletステップの作成（４）-1

6. 作成したsleepybatchlet.class を開いて以下のソースコードに書き換えます。

```
package com.ibm.ws.jbatch.sample.sleepybatchlet;

import java.util.logging.Logger;
import javax.batch.api.BatchProperty;
import javax.batch.api.Batchlet;
import javax.inject.Inject;

public class SleepyBatchlet implements Batchlet {
    /**
     * Default constructor.
     */
    public SleepyBatchlet() {
        // Blank
    }
    private final static Logger logger = Logger.getLogger(SleepyBatchlet.class.getName());
    ....
}
```

6.3. 手順解説 - 7. batchletステップの作成（４）-2

```
.....  
/**  
 * Logging helper.  
 */  
protected static void log(String method, Object msg) {  
    System.out.println("SleepyBatchlet: " + method + ": " + String.valueOf(msg));  
    logger.info("SleepyBatchlet: " + method + ": " + String.valueOf(msg));  
}  
/**  
 * This flag gets set if the batchlet is stopped. This will break the batchlet  
 * out of its sleepy loop.  
 */  
private boolean stopRequested = false;  
/**  
 * The total sleep time, in seconds.  
 */  
.....
```

6.3. 手順解説 - 7. batchletステップの作成 (4) -3

```
.....  
@Inject  
@BatchProperty(name = "sleep.time.seconds")  
String sleepTimeSecondsProperty;  
private int sleepTime_s = 15;  
/**  
 * Main entry point.  
 */  
@Override  
public String process() throws Exception {  
    log("process", "entry");  
    if (sleepTimeSecondsProperty != null) {  
        sleepTime_s = Integer.parseInt(sleepTimeSecondsProperty);  
    }  
    log("process", "sleep for: " + sleepTime_s );  
    int i;  
    for (i = 0; i < sleepTime_s && !stopRequested; ++i) {  
.....
```

batchletの処理を実装します。

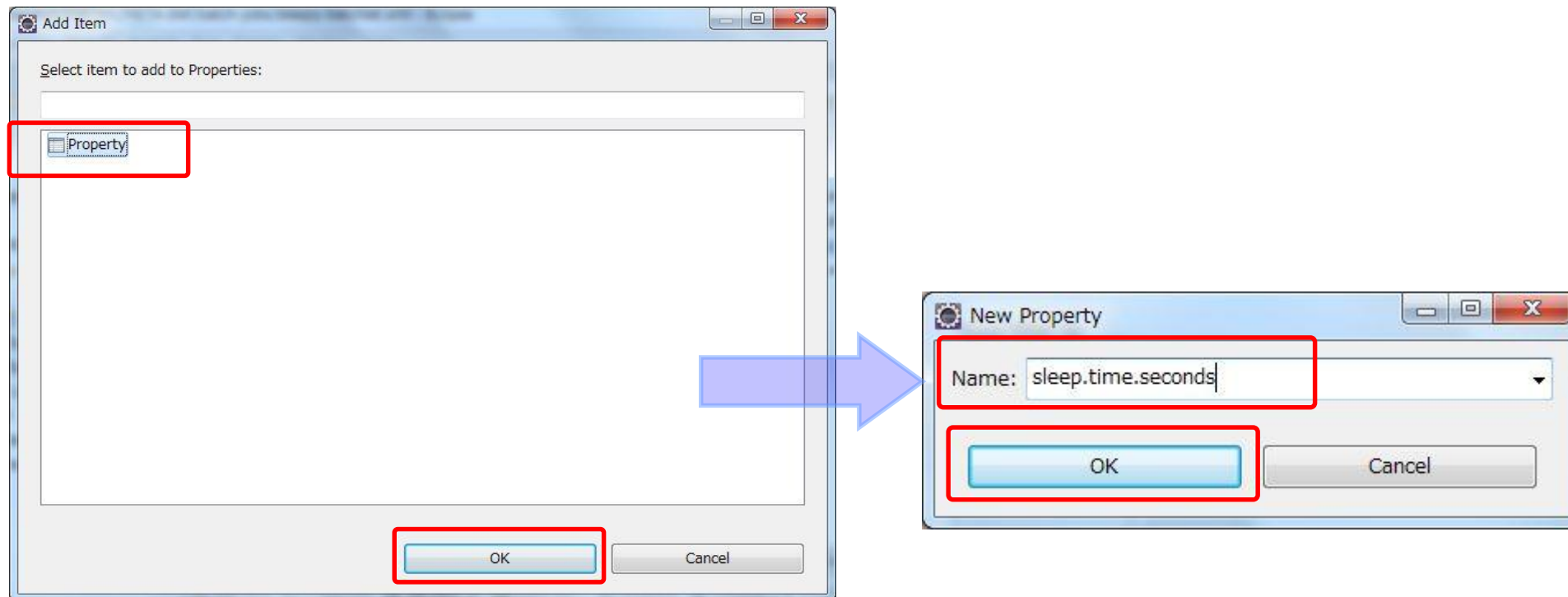
6.3. 手順解説 - 7. batchletステップの作成 (4) -4

```
.....
    log("process", "[" + i + "] sleeping for a second...");
    Thread.sleep(1 * 1000);
}
    String exitStatus = "SleepyBatchlet:i=" + i + ";stopRequested=" + stopRequested;
    log("process", "exit. exitStatus: " + exitStatus);
    return exitStatus;
}
/**
 * Called if the batchlet is stopped by the container.
 */
@Override
public void stop() throws Exception {
    log("stop:", "");
    stopRequested = true;
}
}
```

process() の実行中に、ジョブの中断
が指示された場合に呼び出されます。

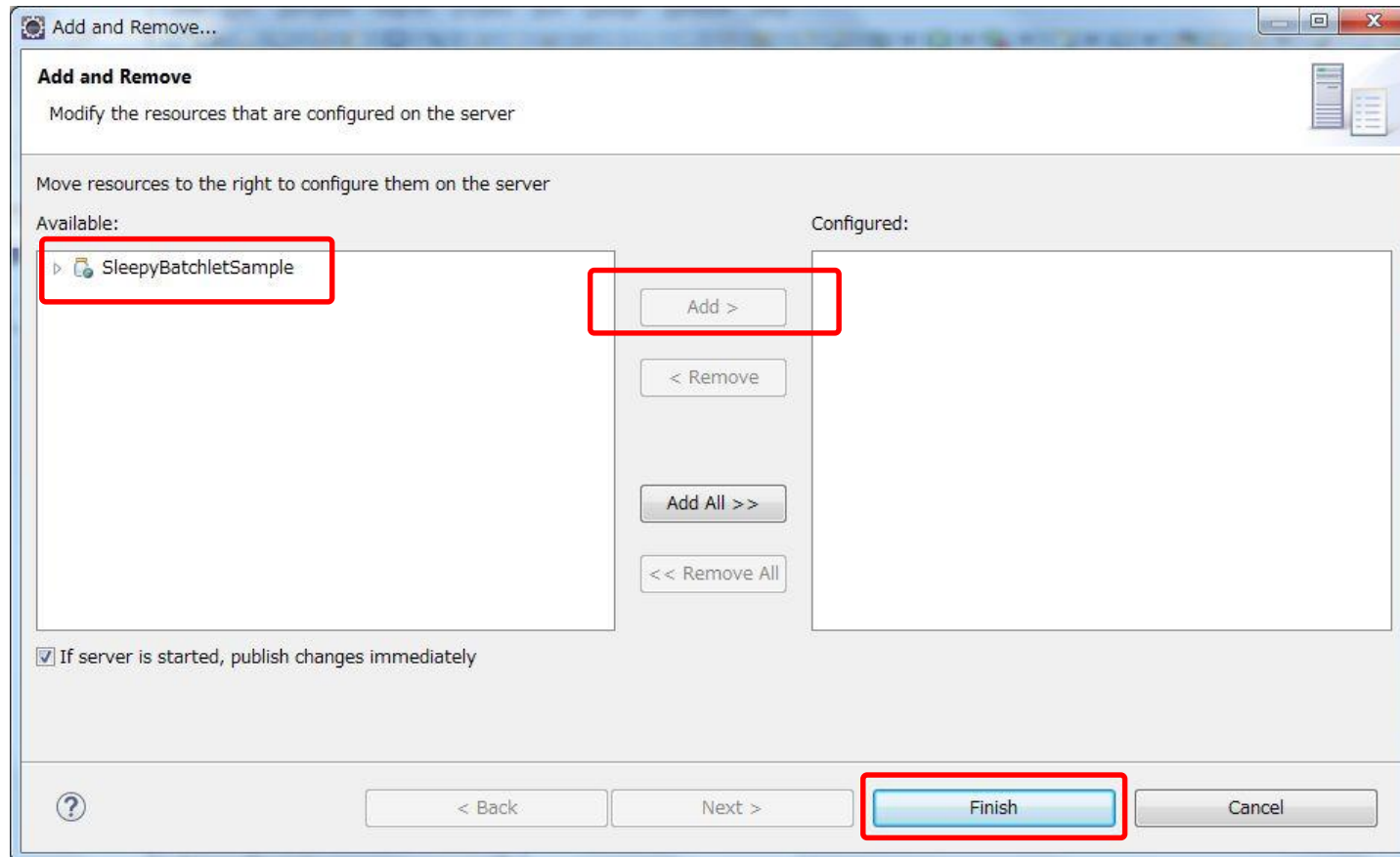
6.3. 手順解説 - 7. batchletステップの作成（5）

1. JSL Editor の「Batchlet()」を選択して、「Add」->「properties」を選択して「OK」をクリックします。
2. JSL Editor の「Properties」を選択して、「Add」->「property」を選択して「sleep.time.seconds」を入力して「OK」をクリックします。Value に"`# {jobParameters['sleep.time.seconds']}`"を入力して `sleepy-batchlet.xml` を保存します。



6.3. 手順解説 - 8. デプロイ

1. Server ビューでWLP を右クリックして「Add and Remove」をクリックします。
2. 「SleepyBatchletSample」を選択して「Add」をクリックします。「Finish」をクリックします。



6.3. 手順解説 - 9. ジョブの実行（1）

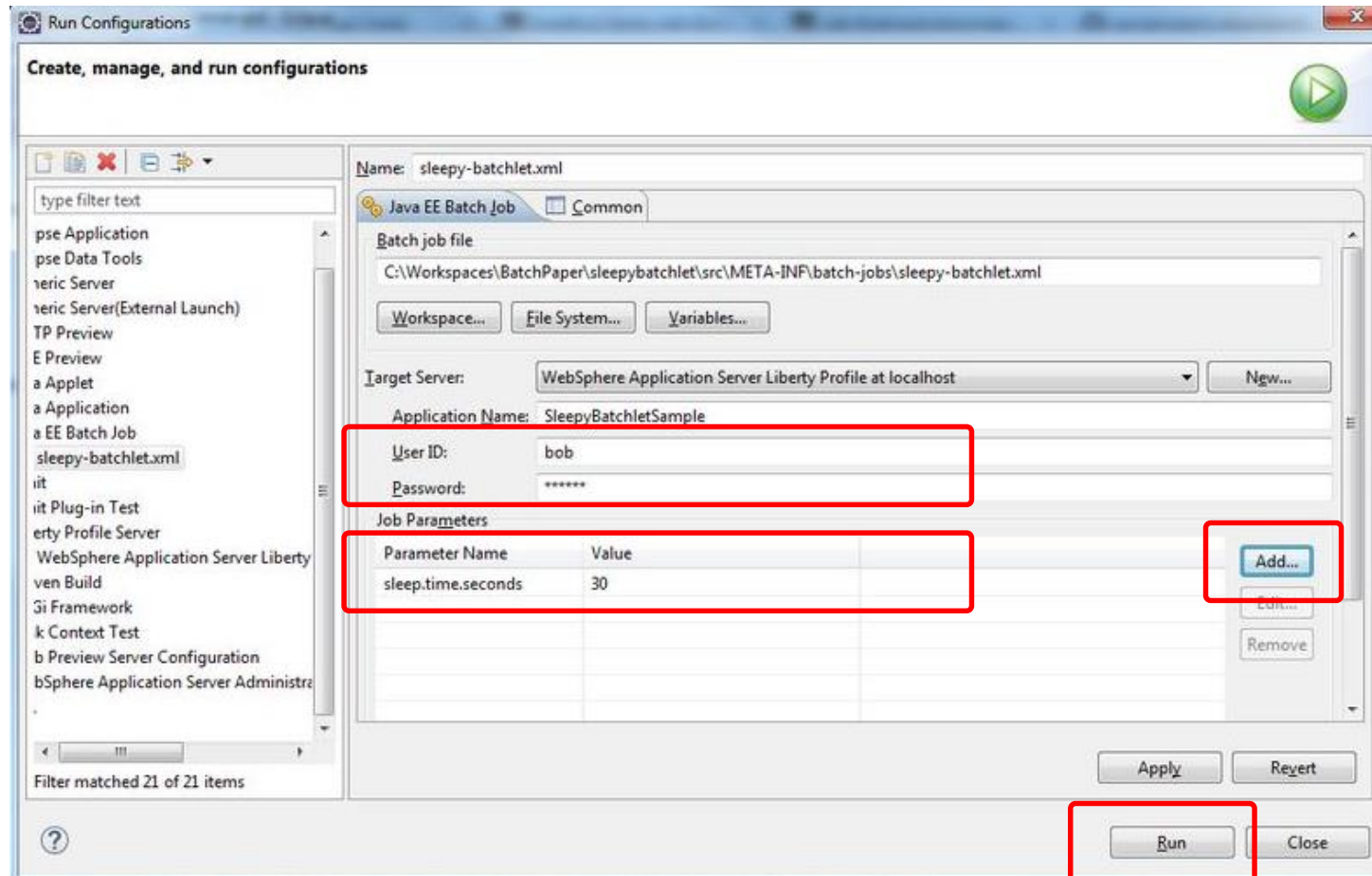
1. server.xml に以下を追加します。server.xml ファイル内で SSL 証明書およびユーザー・レジストリーを作成して、batchManagement-1.0 が自動的に SSL フィーチャーを有効にするようにします。
 - (参考) http://www-01.ibm.com/support/knowledgecenter/SSD28V_8.5.5/com.ibm.websphere.wlp.nd.multiplatform.doc/ae/twlp_batch_configrest.html

server.xml

```
20
21<!-- The default self-signed SSL certificate in this example
22 is intended only for development use and not for production. -->
23
24 <keyStore id="defaultKeyStore" password="Liberty"/>
25<basicRegistry id="basic" realm="ibm/api">
26     <user name="bob" password="bobpwd" />
27 </basicRegistry>
28
```


6.3. 手順解説 - 9. ジョブの実行（2）

2. Workspace のsleepy-batchlet.xml を右クリックして「Run As」 -> 「Java EE Batch Job」をクリックします。

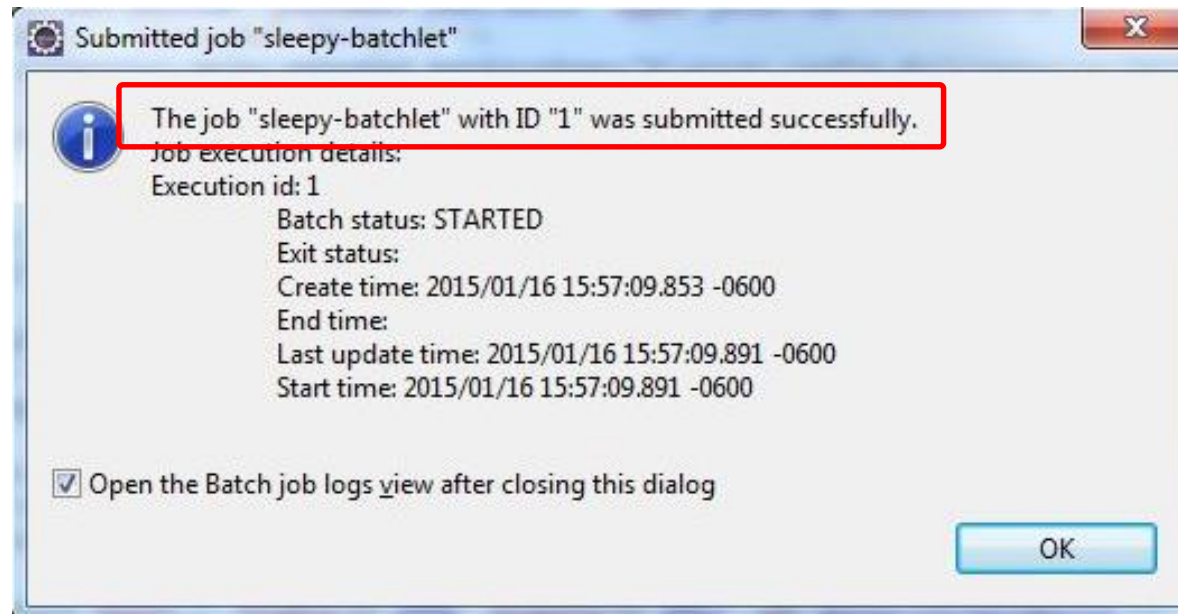


3. Run Configurations に以下を入力して「Run」をクリックします。

- User ID : bob
- Password : bobpwd
- Job Parameters
 - Name : sleep.time.seconds
 - Value : 30

6.3. 手順解説 - 10. ジョブの実行結果

- 前頁の手順でバッチジョブを実行すると、以下のダイアログが出力されます。
- ジョブが正常に登録されたことを確認します。ダイアログに"The job "sleepy-batchlet" with ID "1" was submitted successfully." と表示されていればジョブは正常に登録されています。



- 当資料のjBatch アプリケーションの作成の手順は以下の資料を参考にしています。
 - (参考)WASdev : Creating a simple Java batch application using WebSphere Developer Tools
 - <https://developer.ibm.com/wasdev/docs/creating-simple-java-batch-application-using-websphere-developer-tools/>

参考資料

- JSR-000352 Batch Applications for the Java™ Platform 1.0,
 - http://download.oracle.com/otndocs/jcp/batch-1_0_revA-mrel-spec/index.html
- wasdev : batch タグ アーカイブ
 - <https://developer.ibm.com/wasdev/blog/tag/batch/>
- Knowledge Center : Liberty プロファイルの Java バッチ・アプリケーションのデプロイ
 - http://www-01.ibm.com/support/knowledgecenter/SSD28V_8.5.5/com.ibm.websphere.wlp.core.doc/ae/twlp_container_batch.html

END