

# TERASOLUNA Batch Framework for Java (5.x) Development Guideline

NTT DATA Corporation.

Version 5.0.0.RELEASE, 2017-3-17

# Table of Contents

1.はじめに .....	1
1.1. 利用規約.....	1
1.2. 導入 .....	2
1.2.1. ガイドラインの目的.....	2
1.2.2. ガイドラインの対象読者.....	2
1.2.3. ガイドラインの構成 .....	2
1.2.4. ガイドラインの読み方 .....	3
1.2.4.1. ガイドラインの表記 .....	3
1.2.5. ガイドラインの動作検証環境 .....	4
1.3. 更新履歴.....	5
2. TERASOLUNA Batch Framework for Java (5.x)のコンセプト .....	6
2.1. 一般的なバッチ処理.....	6
2.1.1. 一般的なバッチ処理とは .....	6
2.1.2. バッチ処理に求められる要件 .....	7
2.1.3. バッチ処理で考慮する原則と注意点 .....	8
2.2. TERASOLUNA Batch Framework for Java (5.x)のスタック .....	10
2.2.1. 概要.....	10
2.2.2. TERASOLUNA Batch Framework for Java (5.x)のスタック .....	10
2.2.2.1. 利用するOSSのバージョン .....	10
2.2.3. TERASOLUNA Batch Framework for Java (5.x)の構成要素 .....	12
2.2.3.1. TERASOLUNA Batch Framework for Java (5.x)が実装を提供する機能 .....	14
2.3. Spring Batchのアーキテクチャ.....	16
2.3.1. Overview .....	16
2.3.1.1. Spring Batchとは .....	16
2.3.1.2. Hello, Spring Batch ! .....	16
2.3.1.3. Spring Batchの基本構造 .....	16
2.3.2. Architecture .....	18
2.3.2.1. 処理全体の流れ .....	18
2.3.2.2. Jobの起動 .....	20
2.3.2.3. ビジネスロジックの実行 .....	22
2.3.2.3.1. チャンクモデル .....	22
2.3.2.3.2. タスクレットモデル .....	26
2.3.2.4. JobRepositoryのメタデータスキーマ .....	27
2.3.2.4.1. バージョン .....	28
2.3.2.4.2. ID(シーケンス)定義 .....	28
2.3.2.4.3. テーブル定義 .....	29
2.3.2.4.4. DDLスクリプト .....	32
2.3.2.5. 代表的な性能チューニングポイント .....	32

2.4. TERASOLUNA Batch Framework for Java (5.x)のアーキテクチャ .....	34
2.4.1. 概要 .....	34
2.4.2. ジョブの構成要素 .....	34
2.4.2.1. ジョブ .....	35
2.4.2.2. ステップ .....	35
2.4.3. ステップの実装方式 .....	36
2.4.3.1. チャンクモデル .....	36
2.4.3.2. タスクレットモデル .....	36
2.4.3.3. チャンクモデルとタスクレットモデルの機能差 .....	37
2.4.4. ジョブの起動方式 .....	37
2.4.4.1. 同期実行方式 .....	37
2.4.4.2. 非同期実行方式 .....	38
2.4.4.2.1. 非同期実行方式(DBポーリング) .....	38
2.4.4.2.2. 非同期実行方式(Webコンテナ) .....	39
2.4.5. 利用する際の検討ポイント .....	39
3. アプリケーション開発の流れ .....	41
3.1. バッチアプリケーションの開発 .....	41
3.1.1. ブランクプロジェクトとは .....	41
3.1.2. プロジェクトの作成 .....	41
3.1.3. プロジェクトの構成 .....	45
3.1.4. 開発の流れ .....	47
3.1.4.1. IDEへの取り込み .....	48
3.1.4.2. アプリケーション全体の設定 .....	48
3.1.4.2.1. pom.xmlのプロジェクト情報 .....	48
3.1.4.2.2. データベース関連の設定 .....	48
3.1.5. ジョブの作成 .....	51
3.1.6. プロジェクトのビルドと実行 .....	52
3.1.6.1. アプリケーションのビルド .....	52
3.1.6.2. 環境に応じた設定ファイルの切替 .....	52
3.1.6.2.1. アプリケーションの実行 .....	55
3.2. チャンクモデルジョブの作成 .....	57
3.2.1. Overview .....	57
3.2.1.1. 構成要素 .....	57
3.2.2. How to use .....	57
3.2.2.1. ジョブの設定 .....	58
3.2.2.2. コンポーネントの実装 .....	60
3.2.2.2.1. ItemProcessorの実装 .....	61
3.3. タスクレットモデルジョブの作成 .....	64
3.3.1. Overview .....	64
3.3.1.1. 構成要素 .....	64
3.3.2. How to use .....	64

3.3.2.1. ジョブの設定 .....	64
3.3.2.2. Taskletの実装 .....	66
3.3.2.3. シンプルなTaskletの実装 .....	66
3.3.2.4. チャンクモデルのコンポーネントを利用するTasklet実装 .....	67
3.4. チャンクモデルとタスクレットモデルの使い分け .....	75
4. ジョブの起動 .....	77
4.1. 同期実行 .....	77
4.1.1. Overview .....	77
4.1.2. How to use .....	77
4.1.2.1. 実行方法 .....	78
4.1.2.2. 任意オプション .....	79
4.2. ジョブの起動パラメータ .....	80
4.2.1. Overview .....	80
4.2.2. How to use .....	80
4.2.2.1. パラメータ変換クラスについて .....	80
4.2.2.2. コマンドライン引数から与える .....	81
4.2.2.3. ファイルから標準入力ヘリダイレクトする .....	83
4.2.2.4. パラメータのデフォルト値を設定する .....	84
4.2.2.5. パラメータの妥当性検証 .....	85
4.2.2.5.1. 簡易な妥当性検証 .....	86
4.2.2.5.2. 複雑な妥当性検証 .....	88
4.2.3. How to extends .....	90
4.2.3.1. パラメータとプロパティの併用 .....	90
4.3. 非同期実行(DBポーリング) .....	94
4.3.1. Overview .....	94
4.3.1.1. DBポーリングによるジョブの非同期実行とは .....	94
4.3.1.1.1. TERASOLUNA Batch 5.xが提供する機能 .....	94
4.3.1.1.2. 活用シーン .....	95
4.3.2. Architecture .....	95
4.3.2.1. DBポーリングの処理シーケンス .....	95
4.3.2.2. ポーリングするテーブルについて .....	97
4.3.2.2.1. ジョブ要求テーブルの構造 .....	97
4.3.2.2.2. ジョブ要求シーケンスの構造 .....	98
4.3.2.2.3. ポーリングステータス(polling_status)の遷移パターン .....	98
4.3.2.3. ジョブの起動について .....	99
4.3.2.4. DBポーリング処理で異常が発生した場合について .....	99
4.3.2.4.1. データベース接続障害 .....	99
4.3.2.4.2. 非同期バッチデーモンのプロセス異常終了 .....	100
4.3.2.5. DBポーリング処理の停止について .....	100
4.3.2.6. 非同期実行特有のアプリケーション構成となる点について .....	100
4.3.2.6.1. ApplicationContextの構成 .....	100

4.3.2.6.2. Bean定義の構成 .....	101
4.3.3. How to use .....	101
4.3.3.1. 各種設定 .....	101
4.3.3.1.1. ポーリング処理の設定 .....	101
4.3.3.1.2. ジョブの設定 .....	103
4.3.3.2. 非同期処理の起動から終了まで .....	105
4.3.3.2.1. 非同期バッチデーモンの起動 .....	105
4.3.3.2.2. ジョブの要求 .....	105
4.3.3.2.3. 非同期バッチデーモンの停止 .....	106
4.3.3.3. ジョブのステータス確認 .....	106
4.3.3.4. ジョブが異常終了した後のリカバリ .....	106
4.3.3.4.1. リラン .....	106
4.3.3.4.2. リスタート .....	107
4.3.3.4.3. 停止 .....	107
4.3.3.5. 環境配備について .....	107
4.3.3.6. 累積データの退避について .....	107
4.3.4. How to extend .....	108
4.3.4.1. ジョブ要求テーブルのカスタマイズ .....	108
4.3.4.1.1. 優先度カラムによるジョブ実行順序の制御の例 .....	109
4.3.4.1.2. グループIDによる複数プロセスによる分散処理 .....	111
4.3.4.2. 複数起動 .....	114
4.3.5. Appendix .....	115
4.3.5.1. ジョブ定義のモジュール化について .....	115
4.4. 非同期実行(Webコンテナ) .....	119
4.4.1. Overview .....	119
4.4.2. Architecture .....	119
4.4.2.1. ジョブ起動時における異常発生の検知について .....	121
4.4.2.2. 非同期実行(Webコンテナ)のアプリケーション構成 .....	122
4.4.2.2.1. ApplicationContextの構成 .....	122
4.4.3. How to use .....	124
4.4.3.1. 非同期実行(Webコンテナ)によるアプリケーションの実装概要 .....	124
4.4.3.2. 各種設定 .....	126
4.4.3.3. Webアプリケーションの実装 .....	128
4.4.3.3.1. Webアプリケーションの設定 .....	128
4.4.3.3.2. コントローラで使用するJavaBeansの実装 .....	132
4.4.3.3.3. コントローラの実装 .....	134
4.4.3.3.4. Web/バッチアプリケーションモジュール設定の統合 .....	136
4.4.3.3.5. ビルド .....	137
4.4.3.3.6. デプロイ .....	138
4.4.3.4. REST Clientによるジョブの起動と実行結果確認 .....	138
4.4.4. How to extend .....	140

4.4.4.1. ジョブの停止とリスタート .....	140
4.4.4.2. 複数起動 .....	142
4.5. リスナー .....	143
4.5.1. Overview .....	143
4.5.1.1. リスナーの種類 .....	143
4.5.1.1.1. JobListener .....	143
4.5.1.1.2. StepListener .....	143
4.5.2. How to use .....	145
4.5.2.1. リスナーの実装 .....	145
4.5.2.1.1. インターフェースを実装する場合 .....	146
4.5.2.1.2. アノテーションを付与する場合 .....	148
4.5.2.2. リスナーの設定 .....	150
4.5.2.2.1. 複数リスナーの設定 .....	151
4.5.2.3. インターフェースとアノテーションの使い分け .....	153
5. データの入出力 .....	154
5.1. トランザクション制御 .....	154
5.1.1. Overview .....	154
5.1.1.1. 一般的なバッチ処理におけるトランザクション制御のパターンについて .....	154
5.1.1.2. Architecture .....	155
5.1.2.1. Spring Batchにおけるトランザクション制御 .....	155
5.1.2.1.1. チャンクモデルにおけるトランザクション制御の仕組み .....	156
5.1.2.1.2. タスクレットモデルにおけるトランザクション制御の仕組み .....	158
5.1.2.1.3. モデル別トランザクション制御の選定方針 .....	164
5.1.2.2. 起動方式ごとのトランザクション制御の差 .....	164
5.1.2.2.1. DBポーリングのトランザクションについて .....	164
5.1.2.2.2. WebAPサーバ処理のトランザクションについて .....	166
5.1.3. How to use .....	167
5.1.3.1. 単一データソースの場合 .....	168
5.1.3.1.1. トランザクション制御の実施 .....	168
5.1.3.1.2. 非トランザクショナルなデータソースに対する補足 .....	172
5.1.3.2. 複数データソースの場合 .....	173
5.1.3.2.1. 複数データソースからの取得 .....	173
5.1.3.2.2. 複数データソースへの出力(複数ステップ) .....	177
5.1.3.2.3. 複数データソースへの出力(1ステップ) .....	177
5.1.3.3. 中間方式コミットでの注意点 .....	180
5.2. データベースアクセス .....	182
5.2.1. Overview .....	182
5.2.2. How to use .....	182
5.2.2.1. 共通設定 .....	182
5.2.2.1.1. データソースの設定 .....	183
5.2.2.1.2. MyBatisの設定 .....	184

5.2.2.1.3. Mapper XMLの定義.....	186
5.2.2.1.4. MyBatis-Springの設定.....	186
5.2.2.2. ItemReaderにおけるデータベースアクセス .....	187
5.2.2.2.1. MyBatisのItemReader .....	187
5.2.2.3. ItemWriterにおけるデータベースアクセス .....	189
5.2.2.3.1. MyBatisのItemWriter.....	189
5.2.2.4. ItemReader・ItemWriter以外のデータベースアクセス.....	191
5.2.2.4.1. ItemProcessorでのデータベースアクセス .....	192
5.2.2.4.2. Taskletでのデータベースアクセス .....	194
5.2.2.4.3. リスナーでのデータベースアクセス .....	196
5.3. ファイルアクセス.....	200
5.3.1. Overview .....	200
5.3.1.1. 扱えるファイルの種類 .....	200
5.3.1.2. フラットファイルの入出力を行うコンポーネント .....	203
5.3.2. How to use .....	206
5.3.2.1. 可変長レコード .....	206
5.3.2.1.1. 入力 .....	206
5.3.2.1.2. 出力 .....	209
5.3.2.2. 固定長レコード .....	213
5.3.2.2.1. 入力 .....	213
5.3.2.2.2. 出力 .....	218
5.3.2.3. 単一文字列レコード .....	224
5.3.2.3.1. 入力 .....	224
5.3.2.3.2. 出力 .....	225
5.3.2.4. ヘッダとフッタ .....	226
5.3.2.4.1. 入力 .....	226
5.3.2.4.2. 出力 .....	240
5.3.2.5. 複数ファイル .....	243
5.3.2.5.1. 入力 .....	243
5.3.2.5.2. 出力 .....	244
5.3.2.6. コントロールブレイク .....	247
5.3.3. How To Extend .....	251
5.3.3.1. FieldSetMapperの実装 .....	251
5.3.3.2. XMLファイル .....	254
5.3.3.2.1. 入力 .....	255
5.3.3.2.2. 出力 .....	259
5.3.3.3. マルチフォーマット .....	266
5.3.3.3.1. 入力 .....	267
5.3.3.3.2. 出力 .....	272
5.4. 排他制御 .....	278
5.4.1. Overview .....	278

5.4.1.1. 排他制御の必要性 .....	278
5.4.1.2. ファイルの排他制御 .....	278
5.4.1.3. データベースの排他制御 .....	279
5.4.1.4. 排他制御方式の使い分け .....	279
5.4.1.5. 排他制御とコンポーネントの関係 .....	280
5.4.2. How to use .....	281
5.4.2.1. ファイルの排他制御 .....	281
5.4.2.2. データベースの排他制御 .....	284
5.4.2.2.1. 楽観ロック .....	285
5.4.2.2.2. 悲観ロック .....	286
6. 異常系への対応 .....	288
6.1. 入力チェック .....	288
6.1.1. Overview .....	288
6.1.1.1. 入力チェックの分類 .....	288
6.1.1.2. 入力チェックの全体像 .....	289
6.1.2. How to use .....	291
6.1.2.1. 各種設定 .....	292
6.1.2.2. 入力チェックルールの定義 .....	292
6.1.2.3. 入力チェックの実施 .....	293
6.1.2.4. 入力チェックエラーのハンドリング .....	294
6.1.2.4.1. 処理を異常終了する場合 .....	295
6.1.2.4.2. エラーレコードをスキップする場合 .....	298
6.1.2.4.3. 終了コードの設定 .....	299
6.1.2.5. 例外ハンドリング .....	301
6.2.1. Overview .....	301
6.2.1.1. 例外の分類 .....	301
6.2.1.2. 例外の種類 .....	302
6.2.1.2.1. ビジネス例外 .....	302
6.2.1.2.2. 正常稼働時に発生するライブラリ例外 .....	302
6.2.1.2.3. システム例外 .....	303
6.2.1.2.4. 予期しないシステム例外 .....	303
6.2.1.2.5. 致命的なエラー .....	303
6.2.1.2.6. ジョブ要求リクエスト不正エラー .....	304
6.2.1.3. 例外への対応方法 .....	304
6.2.1.3.1. スキップ .....	305
6.2.1.3.2. リトライ .....	305
6.2.1.3.3. 処理中断 .....	305
6.2.2. How to use .....	306
6.2.2.1. ステップ単位の例外ハンドリング .....	306
6.2.2.1.1. ChunkListenerインターフェースによる例外ハンドリング .....	307
6.2.2.1.2. チャンクモデルにおける例外ハンドリング .....	310

6.2.2.1.3. タスクレットモデルにおける例外ハンドリング .....	314
6.2.2.2. ジョブ単位の例外ハンドリング .....	316
6.2.2.3. 処理継続可否の決定 .....	319
6.2.2.3.1. スキップ .....	319
6.2.2.3.2. リトライ .....	328
6.2.2.3.3. 処理中断 .....	331
6.2.3. Appendix .....	332
6.2.3.1. <skippable-exception-classes>を使わない理由について .....	332
6.3. 処理の再実行 .....	335
6.3.1. Overview .....	335
6.3.2. How to use .....	336
6.3.2.1. ジョブのリラン .....	336
6.3.2.2. ジョブのリストア .....	336
6.3.2.3. ステートレスリストア .....	337
6.3.2.4. ステートフルリストア .....	339
7. ジョブの管理 .....	344
7.1. Overview .....	344
7.1.1. ジョブの実行管理とは .....	344
7.1.1.1. Spring Batch が提供する機能 .....	344
7.2. How to use .....	345
7.2.1. ジョブの状態管理 .....	345
7.2.1.1. 状態の永続化 .....	347
7.2.1.2. ジョブの状態・実行結果の確認 .....	348
7.2.1.2.1. クエリを直接発行する .....	348
7.2.1.2.2. <b>JobExplorer</b> を利用する .....	348
7.2.1.3. ジョブの停止 .....	349
7.2.2. 終了コードのカスタマイズ .....	350
7.2.2.1. ジョブの終了コードを変更する。 .....	351
7.2.2.2. 終了コードのマッピングを追加定義する。 .....	352
7.2.3. 二重起動防止 .....	353
7.2.4. ロギング .....	354
7.2.4.1. ログ出力元の明確化 .....	354
7.2.4.2. ログ監視 .....	356
7.2.4.3. ログ出力先 .....	356
7.2.5. メッセージ管理 .....	356
7.3. Appendix. Spring Batch Admin .....	357
8. フロー制御と並列・多重処理 .....	362
8.1. フロー制御 .....	362
8.1.1. Overview .....	362
8.1.2. How to use .....	365
8.1.2.1. シーケンシャルフロー .....	366

8.1.2.2. ステップ間のデータの受け渡し .....	367
8.1.2.2.1. タスクレットモデルを用いたステップ間のデータ受け渡し .....	368
8.1.2.2.2. チャンクモデルを用いたステップ間のデータ受け渡し .....	371
8.1.3. How to extend .....	372
8.1.3.1. 条件分岐 .....	372
8.1.3.2. 停止条件 .....	373
8.2. 並列処理と多重処理 .....	376
8.2.1. Overview .....	376
8.2.1.1. ジョブスケジューラによる並列処理と多重処理 .....	378
8.2.1.1.1. ジョブスケジューラによるジョブの並列化 .....	378
8.2.1.1.2. ジョブスケジューラによるジョブの多重化 .....	378
8.2.2. How to use .....	379
8.2.2.1. Parallel Step (並列処理) .....	379
8.2.2.2. Partitioning Step (多重処理) .....	382
8.2.2.2.1. 分割数が可変の場合 .....	384
8.2.2.2.2. 分割数が固定の場合 .....	390
9. 利用時の注意点 .....	395
9.1. TERASOLUNA Batch 5.xの注意点について .....	395

# Chapter 1. はじめに

## 1.1. 利用規約

本ドキュメントを使用するにあたり、以下の規約に同意していただく必要があります。同意いただけない場合は、本ドキュメント及びその複製物の全てを直ちに消去又は破棄してください。

1. 本ドキュメントの著作権及びその他一切の権利は、NTTデータあるいはNTTデータに権利を許諾する第三者に帰属します。
2. 本ドキュメントの一部または全部を、自らが使用する目的において、複製、翻訳、翻案することができます。ただし本ページの規約全文、およびNTTデータの著作権表示を削除することはできません。
3. 本ドキュメントの一部または全部を、自らが使用する目的において改変したり、本ドキュメントを用いた二次的著作物を作成することができます。ただし、「参考文献：TERASOLUNA Batch Framework for Java (5.x) Development Guideline」あるいは同等の表現を、作成したドキュメント及びその複製物に記載するものとします。
4. 前2項によって作成したドキュメント及びその複製物を、無償の場合に限り、第三者へ提供することができます。
5. NTTデータの書面による承諾を得ることなく、本規約に定められる条件を超えて、本ドキュメント及びその複製物を使用したり、本規約上の権利の全部又は一部を第三者に譲渡したりすることはできません。
6. NTTデータは、本ドキュメントの内容の正確性、使用目的への適合性の保証、使用結果についての的確性や信頼性の保証、及び瑕疵担保義務も含め、直接、間接に被ったいかなる損害に対しても一切の責任を負いません。
7. NTTデータは、本ドキュメントが第三者の著作権、その他如何なる権利も侵害しないことを保証しません。また、著作権、その他の権利侵害を直接又は間接の原因としてなされる如何なる請求（第三者との間の紛争を理由になされる請求を含む。）に関しても、NTTデータは一切の責任を負いません。

本ドキュメントで使用されている各社の会社名及びサービス名、商品名に関する登録商標および商標は、以下の通りです。

- TERASOLUNA は、株式会社NTTデータの登録商標です。
- その他の会社名、製品名は、各社の登録商標または商標です。

## 1.2. 導入

### 1.2.1. ガイドラインの目的

本ガイドラインではSpring Framework、Spring Batch、MyBatisを中心としたフルスタックフレームワークを利用して、保守性の高いバッチアプリケーション開発をするためのベストプラクティスを提供する。

本ガイドラインを読むことで、ソフトウェア開発(主にコーディング)が円滑に進むことを期待する。

### 1.2.2. ガイドラインの対象読者

本ガイドラインはソフトウェア開発経験のあるアーキテクトやプログラマ向けに書かれており、以下の知識があることを前提としている。

- Spring FrameworkのDIやAOPに関する基礎的な知識がある
- Javaを使用してアプリケーションを開発したことがある
- SQLに関する知識がある
- Mavenを使用したことがある

これからJavaを勉強し始めるという人向けではない。

Spring Frameworkに関して、本ドキュメントを読むための基礎知識があるかどうかを測るために [Spring Framework理解度チェックテスト](#) を参照するとよい。この理解度テストが4割回答できない場合は、別途以下のような書籍で学習することを推奨する。

- [Spring徹底入門 \(翔泳社\) \[日本語\]](#)
- [\[改訂新版\] Spring入門——Javaフレームワーク・より良い設計とアーキテクチャ \[日本語\]](#)
- [Pro Spring 4th Edition \(Apress\)](#)

### 1.2.3. ガイドラインの構成

まず、重要なこととして、本ガイドラインは [TERASOLUNA Server Framework for Java \(5.x\) Development Guideline](#) (以降、TERASOLUNA Server 5.x 開発ガイドライン)のサブセットとして位置づけている。出来る限りTERASOLUNA Server 5.x 開発ガイドラインを活用し説明の重複を省くことで、ユーザの学習コスト低減を狙っている。よって随所にTERASOLUNA Server 5.x 開発ガイドラインへの参照を示しているため、両方のガイドを活用しながら開発を進めていってほしい。

#### [TERASOLUNA Batch Framework for Java \(5.x\)のコンセプト](#)

バッチ処理の基本的な考え方、TERASOLUNA Batch Framework for Java (5.x)の基本的な考え方、Spring Batchの概要を説明する。

#### [アプリケーション開発の流れ](#)

TERASOLUNA Batch Framework for Java (5.x)を利用してアプリケーション開発する上で必ず押さえておかなくてはならない知識や作法について説明する。

#### [ジョブの起動](#)

同期実行、非同期実行、起動パラメータといったジョブの起動方法について説明する。

## データの入出力

データベースアクセス、ファイルアクセスといった、各種リソースへの入出力について説明する。

## 異常系への対応

入力チェックや例外ハンドリングといった異常系について説明する。

## ジョブの管理

ジョブの実行管理の方法について説明する。

## フロー制御と並列・多重処理

ジョブを並列処理/分散処理する方法について説明する。

### 1.2.4. ガイドラインの読み方

以下のコンテンツはTERASOLUNA Batch Framework for Java (5.x)を使用するすべての開発者が読むことを強く推奨する。

- [TERASOLUNA Batch Framework for Java \(5.x\)のコンセプト](#)
- [アプリケーション開発の流れ](#)

以下のコンテンツは通常必要となるため、基本的には読んでおくこと。開発対象に応じて、取捨選択するとよい。

- [ジョブの起動](#)
- [データの入出力](#)
- [異常系への対応](#)
- [ジョブの管理](#)

以下のコンテンツは一歩進んだ実装をする際にはじめて参照すれば良い。

- [フロー制御と並列・多重処理](#)

#### 1.2.4.1. ガイドラインの表記

本ガイドラインの表記について、留意事項を述べる。

*Windows*コマンドプロンプトと*Unix*系ターミナルについて

*Windows*と*Unix*系では表記の違いで動作しなくなる場合は併記する。そうでない場合は、*Unix*系の表記で統一する。

プロンプト記号

*Unix*系の\$にて表記する。

プロンプト表記例

```
$ java -version
```

## Bean定義のプロパティとコンストラクタについて

本ガイドラインでは、**p**と**c**のネームスペースを用いた表記とする。ネームスペースを用いることで、Bean定義の記述が簡潔になったり、コンストラクタ引数が明確になる効果がある。

### ネームスペースを利用した記述

```
<bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
    <property name="lineTokenizer">
        <bean
            class="org.terasoluna.batch.item.file.transform.FixedByteLengthLineTokenizer"
            c:ranges="1-6, 7-10, 11-12, 13-22, 23-32"
            c:charset="MS932"
            p:names="branchId,year,month,customerId,amount"/>
    </property>
</bean>
```

参考までに、ネームスペースを利用しない記述を示す。

### ネームスペースを利用しない記述

```
<bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
    <property name="lineTokenizer">
        <bean
            class="org.terasoluna.batch.item.file.transform.FixedByteLengthLineTokenizer">
            <constructor-arg index="0" value="1-6, 7-10, 11-12, 13-22, 23-32"/>
            <constructor-arg index="1" value="MS932"/>
            <property name="names" value="branchId,year,month,customerId,amount"/>
        </property>
    </bean>
```

なお、ユーザに対してネームスペースを用いる記述を強要することはない。あくまで説明を簡潔にするための配慮であると受け止めてほしい。

## 1.2.5. ガイドラインの動作検証環境

本ガイドラインで説明している内容の動作検証環境については、「[テスト済み環境](#)」を参照されたい。

### 1.3. 更新履歴

更新日付	更新箇所	変更内容
2017-03-17	-	5.0.0 RELEASE 版公開

# Chapter 2. TERASOLUNA Batch Framework for Java (5.x)のコンセプト

## 2.1. 一般的なバッチ処理

### 2.1.1. 一般的なバッチ処理とは

一般的に、バッチ処理とは「まとめて一括処理する」ことを指す。

データベースやファイルから大量のレコードを読み込み、処理し、書き出す処理であることが多い。

バッチ処理には以下の特徴があり、オンライン処理と比較して、応答性より処理スループットを優先した処理方式である。

#### バッチ処理の特徴

- データを一定の量でまとめて処理する。
- 処理に一定の順序がある。
- スケジュールに従って実行・管理される。

次にバッチ処理を利用する主な目的を以下に示す。

#### スループットの向上

データをまとめて処理することで、処理のスループットを向上できる。

ファイルやデータベースは、1件ごとにデータを入出力せず、一定件数にまとめることで、I/O待ちのオーバヘッドが劇的に少くなり効率的である。1件ごとのI/O待ちは微々たるものでも、大量データを処理する場合はその累積が致命的な遅延となる。

#### 応答性の確保

オンライン処理の応答性を確保するため、即時処理を行う必要がない処理をバッチ処理に切り出す。たとえば、すぐに処理結果が必要でない場合、オンライン処理で受付まで処理を行い、裏でバッチ処理を行う構成がある。このような処理方式は、[ディレードバッチ](#)、[ディレードオンライン](#)などと呼ばれる。

#### 時間やイベントへの対応

特定の時間やイベントに応じた処理は、バッチ処理として実装することが素直と言える。

たとえば、業務要件により1ヶ月分のデータを翌月第1週の週末に集計する、システム運用ルールに則って週末日曜の午前2時に1週間分の業務データをバックアップする、などである。

#### 外部システムとの連携上の制約

ファイルなど外部システムとのインターフェースが制約となるために、バッチ処理を利用することもある。

外部システムから送付されてきたファイルは、一定期間のデータをまとめたものになる。これを取り込む処理は、オンライン処理よりもバッチ処理が向いている。

バッチ処理を実現するには、さまざまな技術要素を組み合わせることが一般的である。ここでは、主要な技術を紹介する。

## ジョブスケジューラ

バッチ処理の1実行単位をジョブと呼ぶ。これを管理するためのミドルウェアである。

バッチシステムにおいて、ジョブが数個であることは稀であり、通常は数百、ときには数千にいたる場合もある。そのため、ジョブの関連を定義し、実行スケジュールを管理する専用の仕組みが不可欠になる。

## シェルスクリプト

ジョブを実現する方法の1つ。OSやミドルウェアなどに実装されているコマンドを組み合わせて1つの処理を実現する。

手軽に実装できる反面、複雑なビジネスロジックを記述するには向きであるため、ファイルのコピー・バックアップ・テーブルクリアなど主にシンプルな処理に用いる。また、別のプログラミング言語で実装した処理を実行する際に、起動前の設定や実行後の処理だけをシェルスクリプトが担うことが多い。

## プログラミング言語

ジョブを実現する方法の1つ。シェルスクリプトよりも構造化されたコードを記述でき、開発生産性・メンテナンス性・品質などを確保するのに有利である。そのため、比較的複雑なロジックになりやすいファイルやデータベースのデータを加工するようなビジネスロジックの実装によく使われる。

### 2.1.2. バッチ処理に求められる要件

業務処理を実現するために、バッチ処理に求められる要件には以下のようなものがある。

- 性能向上
  - 一定量のデータをまとめて処理できる。
  - ジョブを並列/多重に実行できる。
- 異常発生時のリカバリ
  - 再実行(手動/スケジュール)ができる。
  - 再処理した時に、処理済レコードのスキップして、未処理部分だけを処理できる。
- 多様な起動方式
  - 同期実行ができる。
  - 非同期実行ができる。
    - 実行契機としては、DBポーリング、HTTPリクエスト、などがある。
- さまざまな入出力インターフェース
  - データベース
  - ファイル
    - CSVやTSVなどの可変長
    - 固定長
    - XML

上記の要件について具体的な内容を以下に示す。

#### 大量データを一定のリソースで効率よく処理できる(性能向上)

大量のデータをまとめて処理することで処理時間を短縮する。このとき重要なのは、「一定のリソースで」の部分である。

100万件でも1億件でも、一定のCPUやメモリの使用で処理でき、件数に応じて緩やかにかつリニアに処理時間が伸びるのが理想である。まとめて処理するには、一定件数ごとにトランザクションを開始・終了させ、まとめてI/O入出力しすることで、使用的リソースを平準化させる仕組みが必要となる。

それでも処理しきれない膨大なデータを相手にする場合は、一歩進んでハードウェアリソースを限界まで使い切る仕組みも追加で必要になる。処理対象データを件数やグループで分割して、複数プロセス・複数スレッドによって多重処理する。さらに推し進めて複数マシンによる分散処理をすることもある。リソースを限界まで使い切る際には、I/Oを限りなく低減することがきわめて重要になる。

#### 可能な限り処理を継続する(異常発生時のリカバリ)

大量データを処理するにあたって、入力データが異常な場合や、システム自体に異常が発生した場合の防護策を考えておく必要がある。

大量データは必然的に処理し終わるまでに長時間かかるが、エラー発生後に復旧までの時間が長期化すると、システム運用に大きな影響を及ぼしてしまう。

たとえば、1000万件のデータを処理する場合を考える。999万件目でエラーになり、それまでの処理をすべてやり直すとしたら、運用スケジュールに影響が出てしまうことは明白である。

このような影響を抑えるために、バッチ処理ならではの処理継続性が重要となる。これにはエラーデータをスキップしながら次のデータを処理する仕組み、処理をリストアートする仕組み、可能な限り自動復旧を試みる仕組み、などが必要となる。また、1つのジョブを極力シンプルなつくりにし、再実行を容易にすることも重要である。

#### 実行契機に応じて柔軟に実行できる(多様な起動方式)

時刻を契機とする場合、オンラインや外部システムとの連携を契機とした場合など、さまざまな実行契機に対応するの仕組みが必要になる。同期実行ではジョブスケジューラから定時になったら処理を起動する、非同期実行ではプロセスを常駐させておきイベントに応じて随時バッチ処理を行う、というような様々な仕組みが一般的に知られている。

#### さまざまな入出力インターフェースを扱える(さまざまな入出力インターフェース)

オンラインや外部システムと連携するということは、データベースはもちろん、CSV/XMLといったさまざまなフォーマットのファイルを扱えることが重要となる。さらに、それぞれの入出力形式を透過的に扱える仕組みがあると実装しやすくなり、複数フォーマットへの対応も迅速に行なえるようになる。

### 2.1.3. バッチ処理で考慮する原則と注意点

バッチ処理システムを構築する際に考慮すべき重要な原則、および、いくつかの一般的な考慮事項を示す。

- 単一のバッチ処理は可能な限り簡素化し、複雑な論理構造を避ける。
- 処理とデータは物理的に近い場所におく(処理を実行する場所にデータを保存する)。
- システムリソース(特にI/O)の利用を最小限にし、できるだけインメモリで多くの操作を実行する。
- また、不要な物理I/Oを避けるため、アプリケーションのI/O(SQLなど)を見直す。

複数のジョブで同じ処理を繰り返さない。

- たとえば、集計処理とレポート処理がある場合に、レポート処理で集計処理を再度することは避ける。
- 常にデータの整合性に関しては最悪の事態を想定する。十分なチェックとデータの整合性を維持するために、データの検証を行う。
- バックアップについて十分に検討する。特にシステムが年中無休で実行されている場合は、バックアップの難易度が高くなる。

•

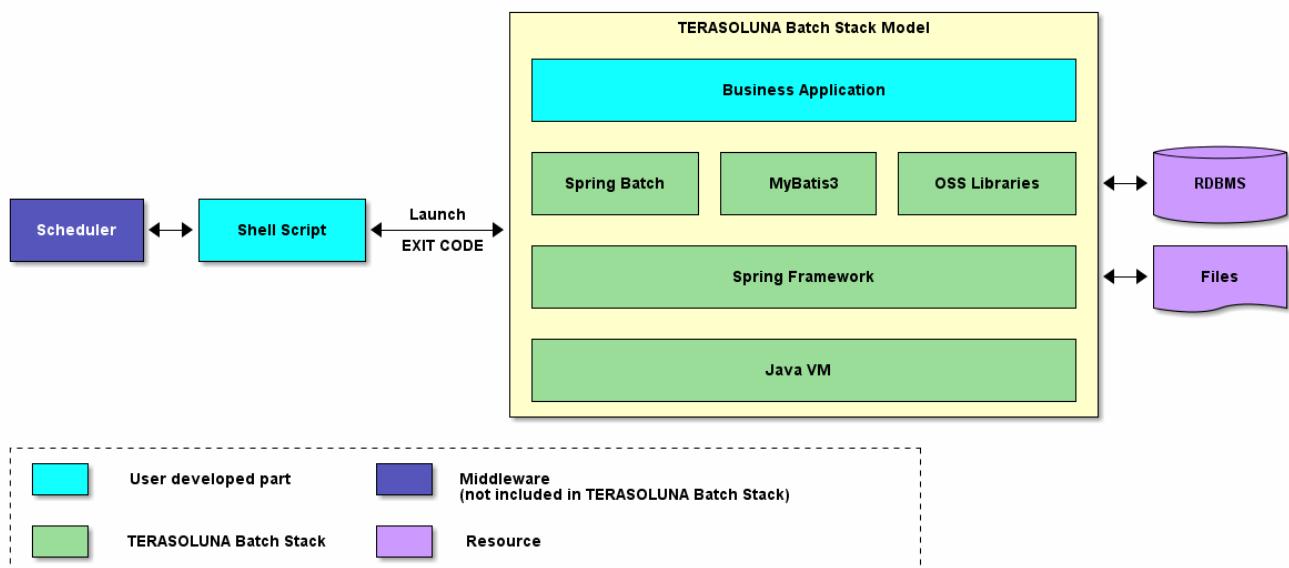
## 2.2. TERASOLUNA Batch Framework for Java (5.x)のスタック

### 2.2.1. 概要

TERASOLUNA Batch Framework for Java (5.x)の構成について説明し、TERASOLUNA Batch Framework for Java (5.x)の担当範囲を示す。

### 2.2.2. TERASOLUNA Batch Framework for Java (5.x)のスタック

TERASOLUNA Batch Framework for Java (5.x)で使用するSoftware Frameworkは、[Spring Framework \(Spring Batch\)](#)を中心としたOSSの組み合わせである。以下にTERASOLUNA Batch Framework for Java (5.x)のスタック概略図を示す。



TERASOLUNA Batch Framework for Java (5.x)のスタック概略図

ジョブスケジューラやデータベースなどの製品についての説明は、本ガイドラインの説明対象外とする。

#### 2.2.2.1. 利用するOSSのバージョン

TERASOLUNA Batch Framework for Java (5.x)のバージョン{revnumber-index}で利用するOSSのバージョン一覧を以下に示す。



TERASOLUNA Batch Framework for Java (5.x)で使用するOSSのバージョンは、原則として、Spring IO platformの定義に準じている。なお、バージョン{revnumber-index}におけるSpring IO platformのバージョンは、[Athens-SR2](#)である。Spring IO platformの詳細については、TERASOLUNA Server Framework for Java (5.x)の[利用するOSSのバージョン](#)を参照。

#### OSS/バージョン一覧

Type	GroupId	ArtifactId	Version	Spring IO platfo rm	Re ma rks
Spring	org.springframework	spring-aop	4.3.5.RELEASE	*	
Spring	org.springframework	spring-beans	4.3.5.RELEASE	*	
Spring	org.springframework	spring-context	4.3.5.RELEASE	*	
Spring	org.springframework	spring-expression	4.3.5.RELEASE	*	
Spring	org.springframework	spring-core	4.3.5.RELEASE	*	
Spring	org.springframework	spring-tx	4.3.5.RELEASE	*	
Spring	org.springframework	spring-jdbc	4.3.5.RELEASE	*	
Spring Batch	org.springframework.batch	spring-batch-core	3.0.7.RELEASE	*	
Spring Batch	org.springframework.batch	spring-batch-infrastructure	3.0.7.RELEASE	*	
Spring Retry	org.springframework.retry	spring-retry	1.1.5.RELEASE	*	
Java Batch	javax.batch	javax.batch-api	1.0.1	*	
Java Batch	com.ibm.jbatch	com.ibm.jbatch-tck-spi	1.0	*	
MyBatis3	org.mybatis	mybatis	3.4.2		
MyBatis3	org.mybatis	mybatis-spring	1.3.1		
MyBatis3	org.mybatis	mybatis-typehandlers-jsr310	1.0.2		
DI	javax.inject	javax.inject	1	*	
ログ出力	ch.qos.logback	logback-classic	1.1.8	*	
ログ出力	ch.qos.logback	logback-core	1.1.8	*	*1
ログ出力	org.slf4j	jcl-over-slf4j	1.7.22	*	
ログ出力	org.slf4j	slf4j-api	1.7.22	*	
入力チェック	javax.validation	validation-api	1.1.0.Final	*	

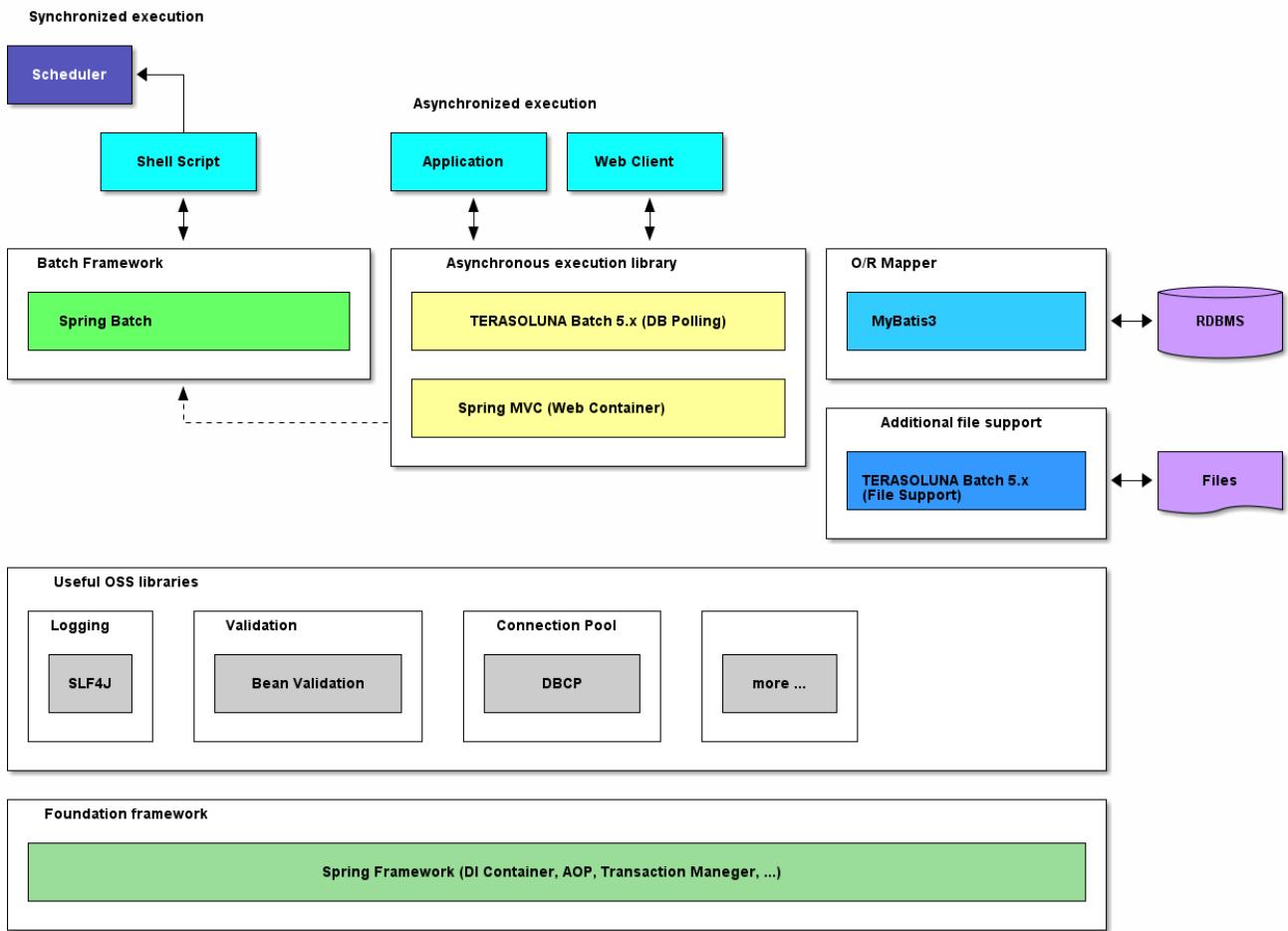
Type	GroupId	ArtifactId	Version	Spring IO plat form	Re ma rks
入力チェック	org.hibernate	hibernate-validator	5.2.4.Final	*	
入力チェック	org.jboss.logging	jboss-logging	3.3.0.Final	*	*1
入力チェック	com.fasterxml	classmate	1.3.3	*	*1
コネクションプール	org.apache.commons	commons-dbcp2	2.1.1	*	
コネクションプール	org.apache.commons	commons-pool2	2.4.2	*	
EL式	org.glassfish	javax.el	3.0.0	*	
インメモリデータベース	com.h2database	h2	1.4.193	*	
XML	com.thoughtworks.xstream	xstream	1.4.9	*	*1
XML	xmlpull	xmlpull	1.1.3.1		*1
XML	xpp	xpp3_min	1.1.4c		*1
XML	xpp	xpp3_min	1.1.4c		*1
JSON	org.codehaus.jettison	jettison	1.2	*	*1

Remarksについて

1. Spring IO platformでサポートしているライブラリが個別に依存しているライブラリ

### 2.2.3. TERASOLUNA Batch Framework for Java (5.x)の構成要素

TERASOLUNA Batch Framework for Java (5.x)のSoftware Framework構成要素について説明する。



Software Framework構成要素の概略図

以下に、各要素の概要を示す。

#### 基盤フレームワーク

フレームワークの基盤として、Spring Frameworkを利用する。DIコンテナをはじめ各種機能を活用する。

- [Spring Framework 4.3](#)

#### バッチフレームワーク

バッチフレームワークとして、Spring Batchを利用する。

- [Spring Batch 3.0](#)

#### 非同期実行

非同期実行を実現する方法として、以下の機能を利用する。

##### DBポーリングによる周期起動

TERASOLUNA Batch Framework for Java (5.x)が提供するライブラリを利用する。

- [非同期実行\(DBポーリング\)](#)

##### Webコンテナ起動

Spring MVCを使用して、Spring Batchと連携をする。

- [Spring MVC 4.3](#)

#### O/R Mapper

MyBatisを利用し、Spring Frameworkとの連携ライブラリとして、MyBatis-Springを使用する。

- [MyBatis 3.4](#)
- [MyBatis-Spring](#)

#### ファイルアクセス

[Spring Batch](#)から提供されている機能に加えて、補助機能をTERASOLUNA Batch Framework for Java (5.x)がする。

- [ファイルアクセス](#)

#### ロギング

ロガーはAPIにSLF4J、実装にLogbackを利用する。

- [SLF4J](#)
- [Logback](#)

#### バリデーション

##### 単項目チェック

単項目チェックにはBean Validationを利用し、実装はHibernate Validatorを使用する。

- [Bean Validation 1.1](#)
- [Hibernate Validator 5.2](#)

##### 相関チェック

相関チェックにはBean Validation、もしくはSpring Validationを利用する。

- [Spring Validation](#)

#### コネクションプール

コネクションプールには、DBCPを利用する。

- [DBCP 2](#)
- [Commons Pool 2](#)

### 2.2.3.1. TERASOLUNA Batch Framework for Java (5.x)が実装を提供する機能

TERASOLUNA Batch Framework for Java (5.x)が実装を提供する機能を以下に示す。

#### TERASOLUNA Batch Framework for Java (5.x)が実装を提供する機能一覧

機能名	概要
<a href="#">非同期実行(DBポーリング)</a>	DBポーリングによる非同期実行を実現する。

	改行なしの固定長ファイルをバイト数で読み込む。
ファイルアクセス	固定長レコードをバイト数で各項目に分解する。
	可変長レコードで囲み文字の出力を制御する。

## 2.3. Spring Batchのアーキテクチャ

### 2.3.1. Overview

TERASOLUNA Server Framework for Java (5.x)の基盤となる、Spring Batchのアーキテクチャについて説明をする。

#### 2.3.1.1. Spring Batchとは

Spring Batchは、その名のとおりバッチアプリケーションフレームワークである。SpringがもつDIコンテナやAOP、トランザクション管理機能をベースとして以下の機能を提供している。

処理の流れを定型化する機能

タスクレットモデル

シンプルな処理

自由に処理を記述する方式である。SQLを1回発行するだけ、コマンドを発行するだけ、といった簡素なケースや複数のデータベースやファイルにアクセスしながら処理するような複雑で定型化しにくいケースで用いる。

チャンクモデル

大量データを効率よく処理

一定件数のデータごとにまとめて入力／加工／出力する方式。データの入力／加工／出力といった処理の流れを定型化し、一部を実装するだけでジョブが実装できる。

様々な起動方法

コマンドライン実行、Servlet上で実行、その他のさまざまな契機での実行を実現する。

様々なデータ形式の入出力

ファイル、データベース、メッセージキューをはじめとするさまざまなデータリソースとの入出力を簡単に行う。

処理の効率化

多重実行、並列実行、条件分岐を設定ベースで行う。

ジョブの管理

実行状況の永続化、データ件数を基準にしたリストアなどを可能にする。

#### 2.3.1.2. Hello, Spring Batch !

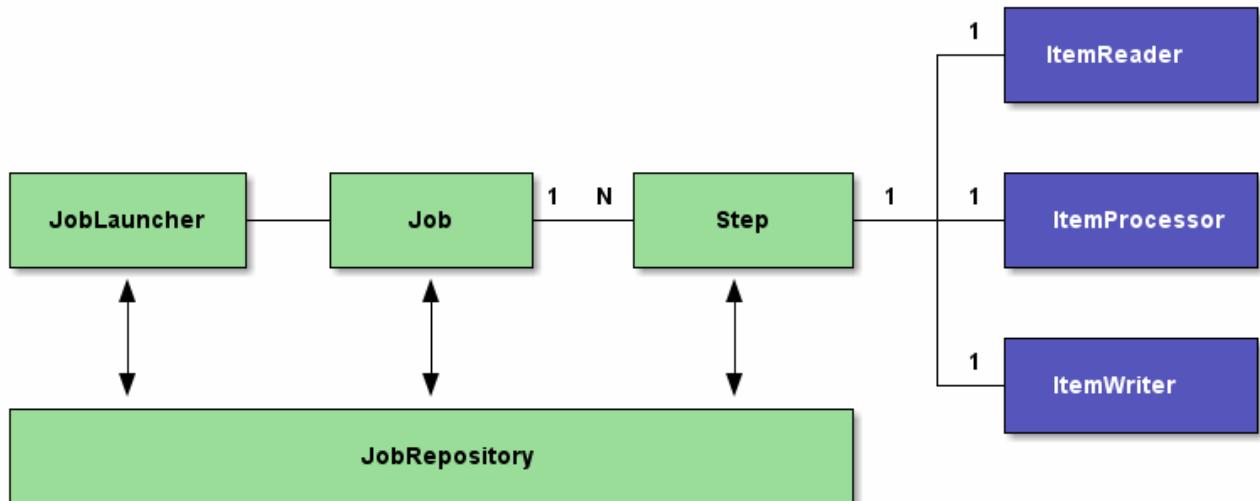
Spring Batchのアーキテクチャを理解する上で、未だSpring Batchに触れたことがない場合は、以下の公式ドキュメントを一読するとよい。Spring Batchを用いた簡単なアプリケーションの作成を通して、イメージを掴んでほしい。

#### [Creating a Batch Service](#)

#### 2.3.1.3. Spring Batchの基本構造

Spring Batchの基本的な構造を説明する。

Spring Batchはバッチ処理の構造を定義している。この構造を理解してから開発を行うことを推奨する。



Spring Batchに登場する主な構成要素

Spring Batchに登場する主な構成要素

構成要素	役割
Job	Spring Batchにおけるバッチアプリケーションの一連の処理をまとめた1実行単位。
Step	Jobを構成する処理の単位。1つのJobに1～N個のStepをもたせることができ。1つのJobを複数のStepに分割して処理することにより、処理の再利用、並列化、条件分岐が可能になる。Stepは、チャンクモデルまたはタスクレットモデル（これらについては後述する）のいずれかで実装する。
JobLauncher	Jobを起動するためのインターフェース。 JobLauncherをユーザが直接利用することも可能だが、javaコマンドからCommandLineJobRunnerを起動することでより簡単にバッチ処理を開始できる。 CommandLineJobRunnerは、JobLauncherを起動するための各種処理を引き受けてくれる。
ItemReader ItemProcessor ItemWriter	チャンクモデルを実装する際に、データの入力／加工／出力の3つに分割するためのインターフェース。 バッチアプリケーションは、この3パターンの処理で構成されることが多いことに由来し、Spring Batchでは主にチャンクモデルでこれらインターフェースの実装を活用する。 ユーザはビジネスロジックをそれぞれの役割に応じて分割して記述する。 データの入出力を担うItemReaderとItemWriterは、データベースやファイルからJavaオブジェクトへの変換、もしくはその逆の処理であることが多い。 そのため、Spring Batchから標準的な実装が提供されている。 ファイルやデータベースからデータの入出力を行う一般的なバッチアプリケーションの場合は、Spring Batchの標準実装をそのまま使用するだけで要件を満たせるケースもある。 データの加工を担うItemProcessorは、入力チェックやビジネスロジックを実装する。タスクレットモデルでは、ItemReader/ItemProcessor/ItemWriterが、1つのTaskletインターフェース実装に置き換わる。Tasklet内に入出力、入力チェック、ビジネスロジックのすべてを実装する必要がある。

構成要素	役割
JobRepository	JobやStepの状況を管理する機構。これらの管理情報は、Spring Batchが規定するテーブルスキーマを元にデータベース上に永続化される。

### 2.3.2. Architecture

[Overview](#)ではSpring Batchの基本構造については簡単に説明した。

これを踏まえて、以下の点について説明をする。

- [処理全体の流れ](#)
- [Jobの起動](#)
- [ビジネスロジックの実行](#)
- [JobRepositoryのメタデータスキーマ](#)

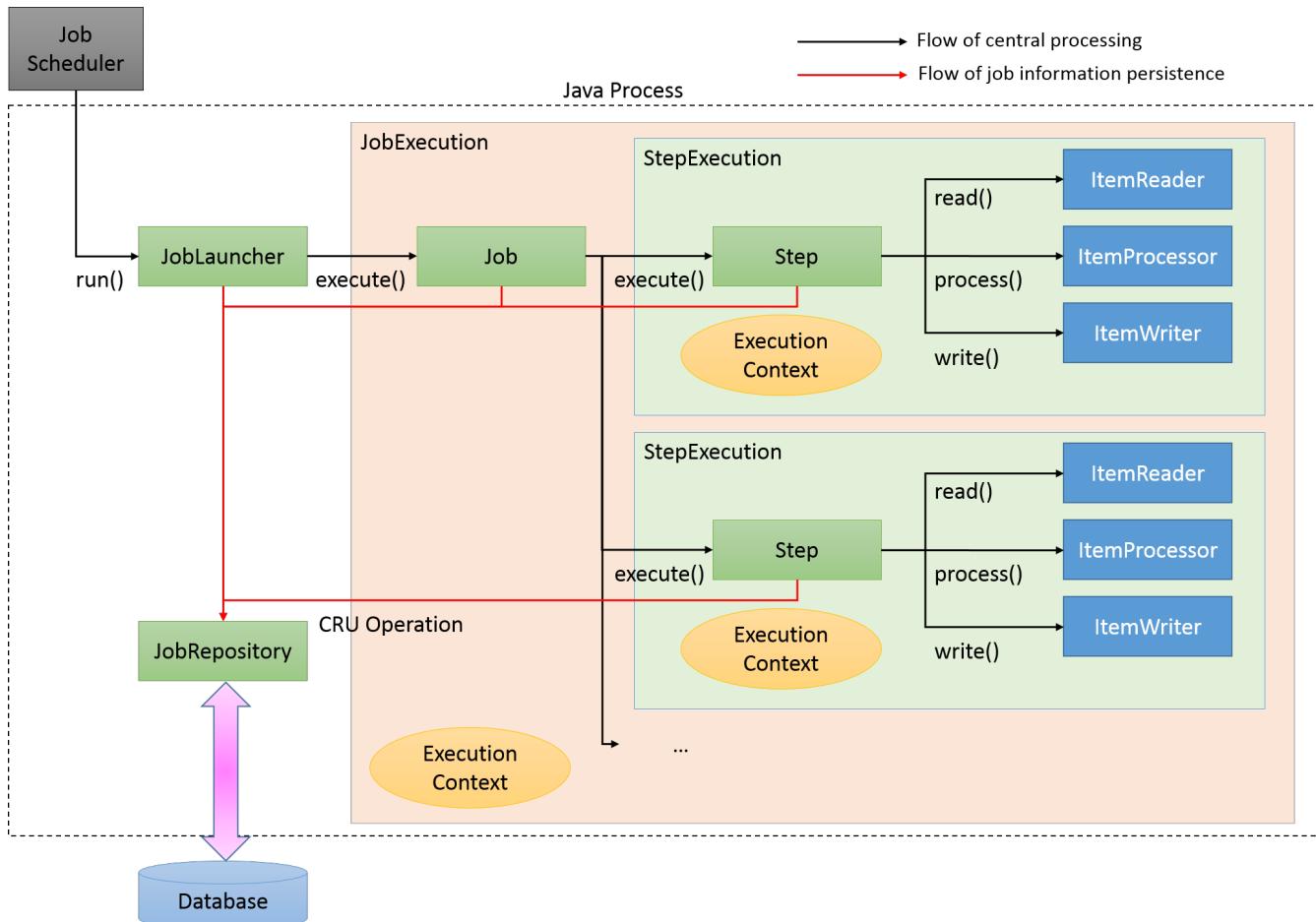
最後に、Spring Batchを利用したバッチアプリケーションの性能チューニングポイントについて説明をする。

- [代表的な性能チューニングポイント](#)

#### 2.3.2.1. 処理全体の流れ

Spring Batchの主な構成要素と処理全体の流れについて説明をする。また、ジョブの実行状況などのメタデータがどのように管理されているかについても説明する。

Spring Batchの主な構成要素と処理全体の流れ(チャネルモデル)を下図に示す。



Spring Batchの主な構成要素と処理全体の流れ

中心的な処理の流れ(黒線)とジョブ情報を永続化する流れ(赤線)について説明する。

#### 中心的な処理の流れ

1. ジョブスケジューラからJobLauncherが起動される。
2. JobLauncherからJobを実行する。
3. JobからStepを実行する。
4. StepはItemReaderによって入力データを取得する。
5. StepはItemProcessorによって入力データを加工する。
6. StepはItemWriterによって加工されたデータを出力する

#### ジョブ情報を永続化する流れ

1. JobLauncherはJobRepositoryを介してDatabaseにJobInstanceを登録する。
2. JobLauncherはJobRepositoryを介してDatabaseにジョブが実行開始したことを登録する。
3. JobStepはJobRepositoryを介してDatabaseに入出力件数や状態など各種情報を更新する。
4. JobLauncherはJobRepositoryを介してDatabaseにジョブが実行終了したことを登録する。

新たに構成要素と永続化に焦点をあてたJobRepositoryについての説明を以下に示す。

#### 永続化に関連する構成要素

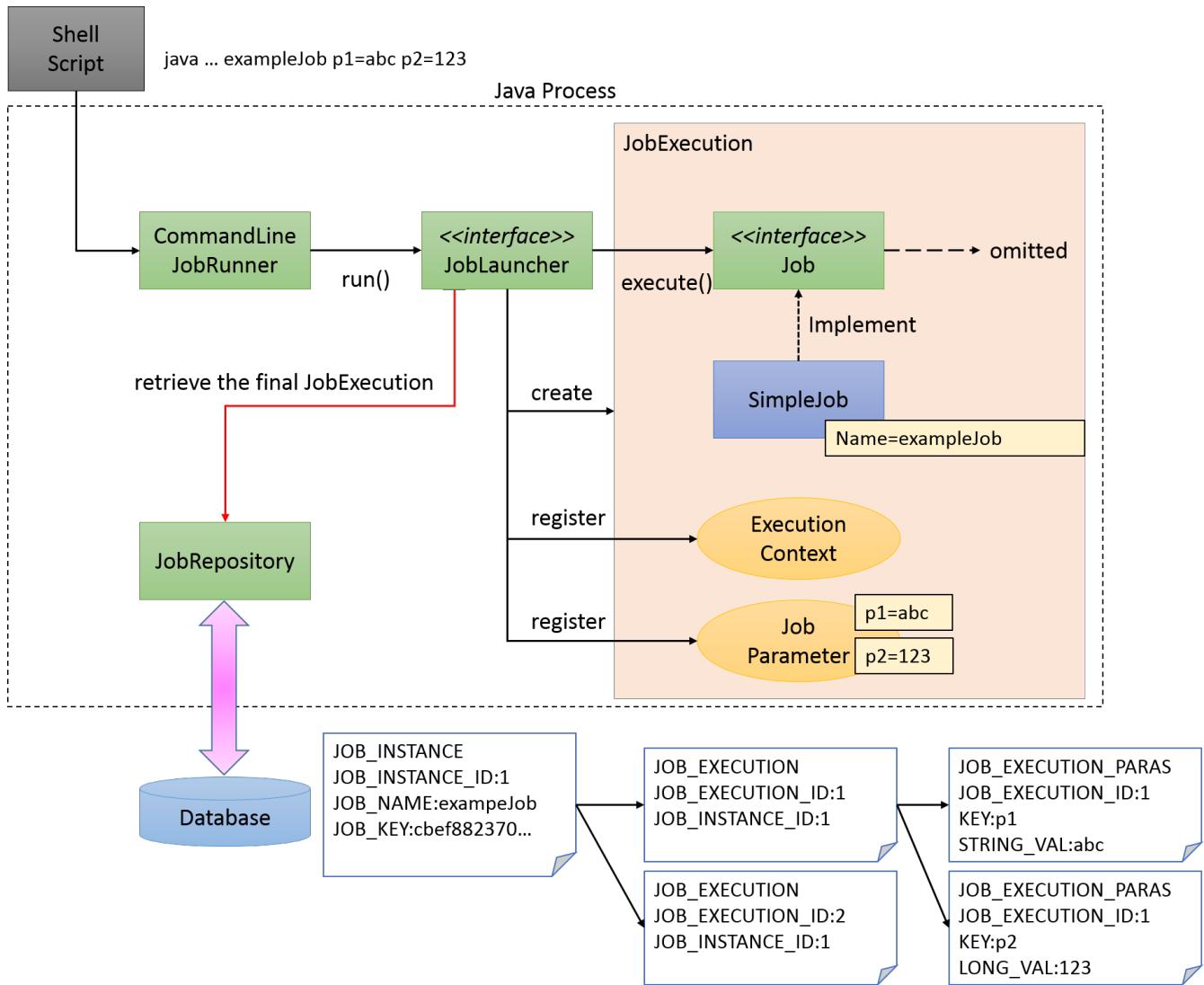
構成要素	役割
JobInstance	<p>Spring BatchはJobの「論理的」な実行を示す。JobInstanceをJob名と引数によって識別している。言い換えると、Job名と引数が同一である実行は、同一JobInstanceの実行と認識し、前回起動時の続きとしてJobを実行する。</p> <p>対象のJobが再実行をサポートしており、前回実行時にエラーなどで処理が途中で中断していた場合は処理の途中から実行される。</p> <p>一方、再実行をサポートしていないJobや、対象のJobInstanceがすでに正常に処理が完了している場合は例外が発生し、Javaプロセスが異常終了する。</p> <p>たとえば、すでに正常に処理が完了している場合はJobInstanceAlreadyCompleteExceptionが発生する。</p>
JobExecution ExecutionContext	<p>JobExecutionはJobの「物理的」な実行を示す。JobInstanceとは異なり、同一のJobを再実行する場合も別のJobExecutionとなる。結果、JobInstanceとJobExecutionは1対多の関係になる。</p> <p>同一のJobExecution内で処理の進捗状況などのメタデータを共有するための領域として、ExecutionContextがある。ExecutionContextは主にSpring Batchがフレームワークの状態などを記録するために使用されているが、アプリケーションがExecutionContextへアクセスする手段も提供されている。</p> <p>JobExecutionContextに格納するオブジェクトは、<a href="#">java.io.Serializable</a>を実装したクラスでなければならない。</p>
StepExecution ExecutionContext	<p>StepExecutionはStepの「物理的」な実行を示す。JobExecutionとStepExecutionは1対多の関係になる。</p> <p>JobExecutionと同様に、Step内でデータを共有するための領域ExecutionContextがある。データの局所化という観点から、複数のStepで共有しなくてもよい情報はJobのExecutionContextを使用するのではなく、対象StepのExecutionContextを利用したほうがよい。</p> <p>StepExecutionContextに格納するオブジェクトは、<a href="#">java.io.Serializable</a>を実装したクラスでなければならない。</p>
JobRepository	<p>JobExecutionやStepExecutionなどのバッチアプリケーション実行結果や状態を管理するためのデータを管理、永続化する機能を提供する。</p> <p>一般的なバッチアプリケーションはJavaプロセスを起動することで処理が開始し、処理の終了とともにJavaプロセスも終了させるケースが多い。</p> <p>そのためこれらのデータはJavaプロセスを跨いで参照される可能性があることから、揮発性なメモリ上だけではなくデータベースなどの永続層へ格納する。</p> <p>データベースに格納する場合は、JobExecutionやStepExecutionを格納するためのテーブルやシーケンスなどのデータベースオブジェクトが必要になる。</p> <p>Spring Batchが提供するスキーマ情報を元にデータベースオブジェクトを生成する必要がある。</p>

Spring Batchが重厚にメタデータの管理を行っている理由は、再実行を実現するためである。バッチ処理を再実行可能にするには、前回実行時のスナップショットを残しておく必要があり、メタデータやJobRepositoryはそのための基盤となっている。

### 2.3.2.2. Jobの起動

Jobをどのように起動するかについて説明する。

Javaプロセス起動直後にバッチ処理を開始し、バッチ処理が完了後にJavaプロセスを終了するケースを考える。下図にJavaプロセス起動からバッチ処理を開始までについて処理の流れを示す。



Javaプロセス起動からバッチ処理を開始までの処理の流れ

#### Javaプロセスの起動とJobの開始

Javaプロセス起動と同時に、Spring Batch上で定義されたJobを開始するためには、Javaを起動するシェルスクリプトを記述するのが一般的である。Spring Batchが提供するCommandLineJobRunnerを使用すると、ユーザが定義したSpring Batch上のJobを簡単に起動することができる。

CommandLineJobRunnerを使用したジョブの起動コマンドは以下のとおりである。

XMLによるBean定義を行った場合の起動コマンド

```
java -cp ${CLASSPATH}
org.springframework.batch.core.launch.support.CommandLineJobRunner <jobPath> <jobName>
<JobArgumentName1>=<value1> <JobArgumentName2>=<value2> ...
```

#### ジョブパラメータの指定

CommandLineJobRunnerは起動するJob名だけでなく、引数(ジョブパラメータ)を渡すことも可能である。引数は前述した例のように、**<Job引数名>=<値>**の形式で指定する。すべての引数はCommandLineJobRunnerやJobLauncherが解釈とチェックを行なったうえで、JobExecutionへJobParametersに変換して格納する。詳細は[ジョブの起動パラメータ](#)を参照のこと。

## JobInstanceの登録と復元

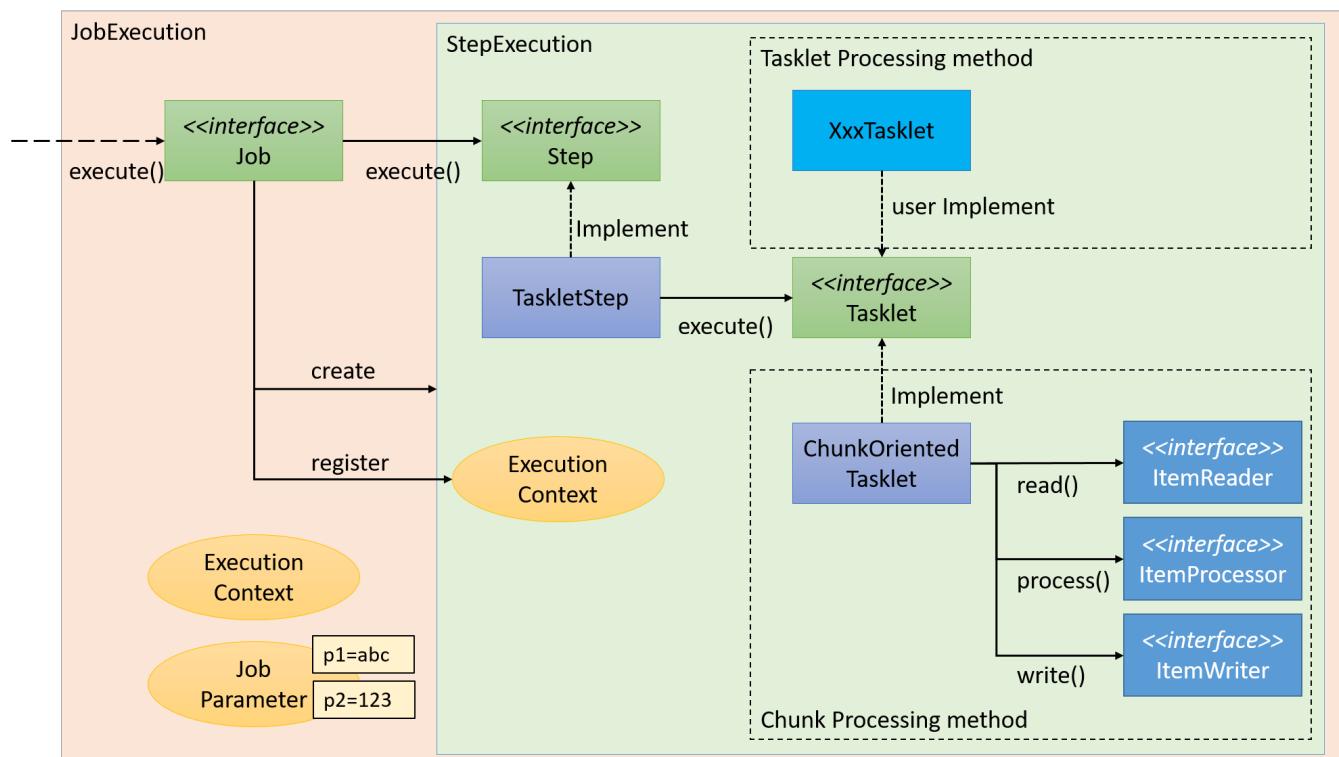
JobLauncherがJobRepositoryからJob名と引数に合致するJobInstanceをデータベースから取得する。

- 該当するJobInstanceが存在しない場合は、JobInstanceを新規登録する。
  - 該当するJobInstanceが存在した場合は、紐付いているJobExecutionを復元する。
    - Spring Batchでは日次実行など繰り返して起動する可能性のあるJobに対しては、JobInstanceがユニークにするためだけの引数を追加する方法がとられている。たとえば、システム時刻であったり、乱数を引数に追加する方法が挙げられる。
- 本ガイドラインで推奨している方法については[パラメータ変換クラスについて](#)を参照。

### 2.3.2.3. ビジネスロジックの実行

Spring Batchでは、JobをStepと呼ぶさらに細かい単位に分割する。Jobが起動すると、Jobは自身に登録されているStepを起動し、StepExecutionを生成する。Stepはあくまで処理を分割するための枠組みであり、ビジネスロジックの実行はStepから呼び出されるTaskletに任せられている。

StepからTaskletへの流れを以下に示す。



### StepからTaskletへの処理の流れ

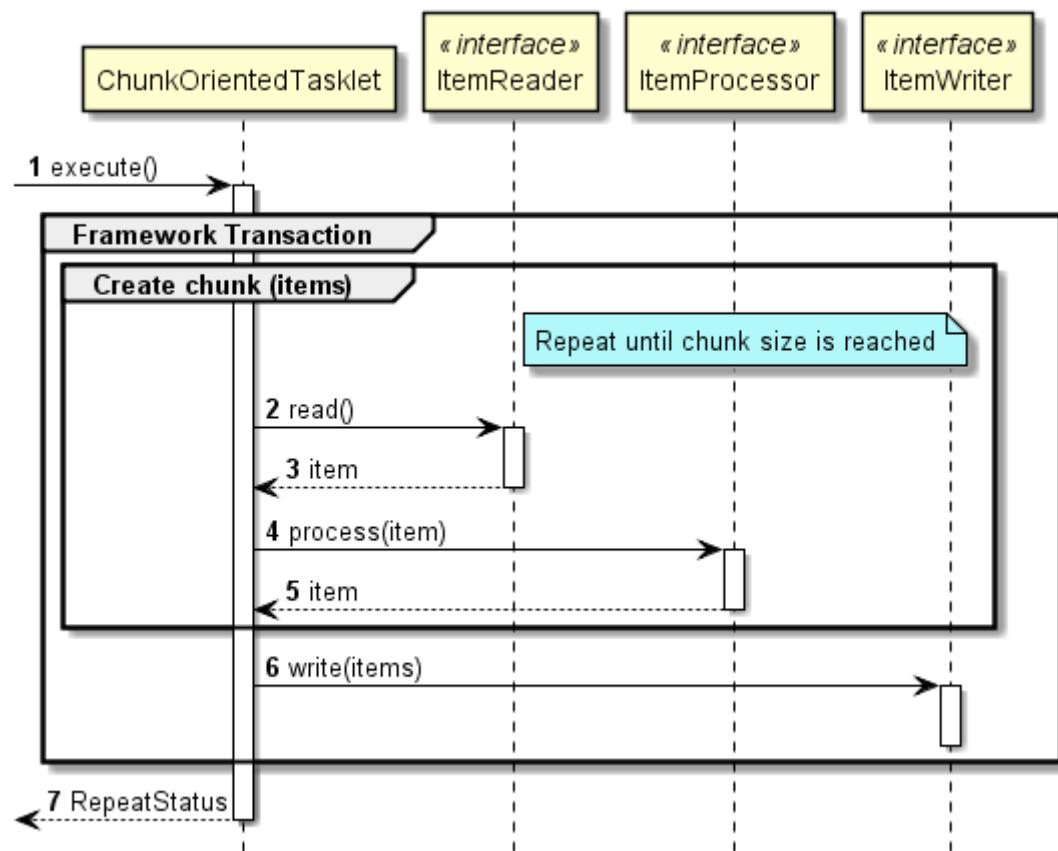
Taskletの実装方法には「チャンクモデル」と「タスクレットモデル」の2つの方式がある。概要についてはすでに説明しているため、ここではその構造について説明する。

#### 2.3.2.3.1. チャンクモデル

前述したようにチャンクモデルとは、処理対象となるデータを1件ずつ処理するのではなく、一定数の塊(チャンク)を単位として処理を行う方式である。ChunkOrientedTaskletがチャンク処理をサポートしたTaskletの具象クラスとなる。このクラスがもつcommit-intervalという設定値により、チャンクに含めるデータの最大件数(以降、「チャンク数」と呼ぶ)を調整することができる。ItemReader、ItemProcessor、ItemWriterは、いずれもチャンク処理を前提としたインターフェースとなっている。

次に、ChunkOrientedTasklet がどのように ItemReader、ItemProcessor、ItemWriter を呼び出しているかを説明する。

ChunkOrientedTasklet が 1 つのチャンクを処理するシーケンス図を以下に示す。



### ChunkOrientedTaskletによるチャンク処理

ChunkOrientedTasklet は、チャンク数分だけ ItemReader および ItemProcessor、すなわちデータの読み込みと加工を繰り返し実行する。チャンク数分のデータすべての読み込みが完了してから、ItemWriter のデータ書き込み処理が 1 回だけ呼び出され、チャンクに含まれるすべての加工済みデータが渡される。データの更新処理がチャンクに対して 1 回呼び出されるように設計されているのは、JDBC の addBatch 、 executeBatch のように I/O をまとめやすくするためである。

次に、チャンク処理において実際の処理を担う ItemReader、ItemProcessor、ItemWriter について紹介する。各インターフェースともユーザが独自に実装を行うことが想定されているが、Spring Batch が提供する汎用的な具象クラスでまかなうことができる場合がある。

特に ItemProcessor はビジネスロジックそのものが記述されることが多いため、Spring Batch からは具象クラスがあまり提供されていない。ビジネスロジックを記述する場合は ItemProcessor インターフェースを実装する。ItemProcessor はタイプセーフなプログラミングが可能になるよう、入出力で使用するオブジェクトの型をそれぞれジェネリクスに指定できるようになっている。

以下に簡単な ItemProcessor の実装例を示す。

## ItemProcessorの実装例

```
public class MyItemProcessor implements  
    ItemProcessor<MyInputObject, MyOutputObject> { // (1)  
    @Override  
    public MyOutputObject process(MyInputObject item) throws Exception { // (2)  
  
        MyOutputObject processedObject = new MyOutputObject(); // (3)  
  
        // Coding business logic for item of input data  
  
        return processedObject; // (4)  
    }  
}
```

項番	説明
(1)	入出力で使用するオブジェクトの型をそれぞれジェネリクスに指定したItemProcessorインターフェースを実装する。
(2)	processメソッドを実装する。引数のitemが入力データである。
(3)	出力オブジェクトを作成し、入力データのitemに対して処理したビジネスロジックの結果を格納する。
(4)	出力オブジェクトを返却する。

ItemReaderやItemWriterは様々な具象クラスがSpring Batchから提供されており、それらを利用する上で十分な場合が多い。しかし、特殊な形式のファイルを入出力したりする場合は、独自のItemReaderやItemWriterを実装した具象クラスを作成し使用することができる。

実際のアプリケーション開発時におけるビジネスロジックの実装に関しては、[アプリケーション開発の流れ](#)を参照。

最後にSpring Batchが提供するItemReader、ItemProcessor、ItemWriterの代表的な具象クラスを示す。

Spring Batchが提供するItemReader、ItemProcessor、ItemWriterの代表的な具象クラス

インターフェース	具象クラス名	概要
ItemReader	FlatFileItemReader	CSVファイルなどの、フラットファイル(非構造的なファイル)の読み込みを行う。Resourceオブジェクトをインプットとし、区切り文字やオブジェクトへのマッピングルールをカスタマイズすることができる。
	StaxEventItemReader	XMLファイルの読み込みを行う。名前とおり、StAXをベースとしたXMLファイルの読み込みを行う実装となっている。
	JdbcPagingItemReader JdbcCursorItemReader	JDBCを使用してSQLを実行し、データベース上のレコードを読み込む。データベース上にある大量のデータを処理する場合は、全件をメモリ上に読み込むことを避け、一度の処理に必要なデータのみの読み込み、破棄を繰り返す必要がある。 JdbcPagingItemReaderはJdbcTemplateを用いてSELECT SQLをページごとに分けて発行することで実現する。一方、JdbcCursorItemReaderはJDBCのカーソルを使用することで、1回のSELECT SQLの発行で実現する。 <b>▲ TERASOLUNA Batch 5.xでは</b> MyBatisを利用するすることを基本とする。
	MyBatisCursorItemReader MyBatisPagingItemReader	MyBatisと連携してデータベース上のレコードを読み込む。MyBatisが提供しているSpring連携ライブラリMyBatis-Springから提供されている。PagingとCursorの違いについては、MyBatisを利用して実現していること以外はJdbcXXXItemReaderと同様。 その他に、JPA実装やHibernateなどと連携してデータベース上のレコードを読み込むJpaPagingItemReader、HibernatePagingItemReader、 HibernateCursorItemReaderが提供されている。 <b>▲ TERASOLUNA Batch 5.xでは</b> MyBatisCursorItemReaderを利用するすることを基本とする。
	JmsItemReader AmqpItemReader	JMSやAMQPからメッセージを受信し、その中に含まれるデータの読み込みを行う。
	PassThroughItemProcessor	何も行なわない。入力データの加工や修正が不要な場合に使用する。
ItemProcessor	ValidatingItemProcessor	入力チェックを行う。入力チェックルールの実装には、Spring Batch独自の org.springframework.batch.item.validator.Validatorを実装する必要がある。 しかし、Springから提供されている汎用的なorg.springframework.validation.ValidatorへのアダプタであるSpringValidatorが提供されており、 org.springframework.validation.Validatorのルールを利用できる。 <b>▲ TERASOLUNA Batch 5.xでは</b> ValidatingItemProcessorの利用は禁止している。 詳細は、 <a href="#">入力チェック</a> を参照。
	CompositeItemProcessor	同一の入力データに対し、複数のItemProcessorを逐次的に実行する。ValidatingItemProcessorによる入力チェックの後にビジネスロジックを実行したい場合などに有効。

インターフェース	具象クラス名	概要
ItemWriter	FlatFileItemWriter	処理済みのJavaオブジェクトを、CSVファイルなどのフラットファイルとして書き込みを行う。区切り文字やオブジェクトからファイル行へのマッピングルールをカスタマイズできる。
	StaxEventItemWriter	処理済みのJavaオブジェクトをXMLファイルとして書き込みを行う。
	JdbcBatchItemWriter	JDBCを使用してSQLを実行し、処理済みのJavaオブジェクトをデータベースへ出力する。内部ではJdbcTemplateが使用されている。
	MyBatisBatchItemWriter	MyBatisと連携して、処理済みのJavaオブジェクトをデータベースへ出力する。MyBatisが提供しているSpring連携ライブラリMyBatis-Springから提供されている。 ▲ TERASOLUNA Batch 5.xでは、JPA実装やHibernate向けのJpaItemWriter、HibernateItemWriterは利用しない。
	JmsItemWriter AmqpItemWriter	処理済みのJavaオブジェクトを、JMSやAMQPでメッセージを送信する。

#### *PassThroughItemProcessor*の省略

XMLでジョブを定義する場合は、ItemProcessorの設定を省略することができる。省略した場合、PassThroughItemProcessorと同様に何もせずに入力データをItemWriterへ受け渡すことになる。

#### *ItemProcessor*の省略



```
<batch:job id="exampleJob">
  <batch:step id="exampleStep">
    <batch:tasklet>
      <batch:chunk reader="reader" writer="writer" commit-
interval="10" />
    </batch:tasklet>
  </batch:step>
</batch:job>
```

#### 2.3.2.3.2. タスクレットモデル

チャンクモデルは、複数の入力データを1件ずつ読み込み、一連の処理を行うバッチアプリケーションに適した枠組みとなっている。しかし、時にはチャンク処理の型に当てはまらないような処理を実装することもある。たとえば、システムコマンドを実行したり、制御用テーブルのレコードを1件だけ更新したいような場合などである。

そのような場合には、チャンク処理によって得られる性能面のメリットが少なく、設計や実装を困難にするデメリットの方が大きいため、タスクレットモデルを使用するほうが合理的である。

タスクレットモデルを使用する場合は、Spring Batchから提供されているTaskletインターフェースをユーザが実装する必要がある。また、Spring Batchでは以下の具象クラスが提供されているが、TERASOLUNA Batch 5.xでは以降説明しない。

## Spring Batchが提供するTaskletの具象クラス

クラス名	概要
SystemCommandTasklet	非同期にシステムコマンドを実行するためのTasklet。commandプロパティに実行したいコマンドを指定する。 システムコマンドは呼び出し元のスレッドと別スレッドで実行されるため、タイムアウトを設定したり、処理中にシステムコマンドの実行スレッドをキャンセルすることも可能である。
MethodInvokingTaskletAdapter	POJOクラスに定義された特定のメソッドを実行するためのTasklet。targetObjectプロパティに対象クラスのBeanを、targetMethodプロパティに実行させたいメソッド名を指定する。 POJOクラスはバッチ処理の終了状態をメソッドの返り値として返却することができるが、その場合は後述するExitStatusをメソッドの返り値とする必要がある。 他の型で返り値を返却した場合は、返り値の内容にかかわらず正常終了した(ExitStatus.COMPLETED)とみなされる。

### 2.3.2.4. JobRepositoryのメタデータスキーマ

JobRepositoryのメタデータスキーマについて説明する。

なお、Spring Batchのリファレンス [Appendix B. Meta-Data Schema](#) にて説明されている内容も含めて、全体像を説明する。

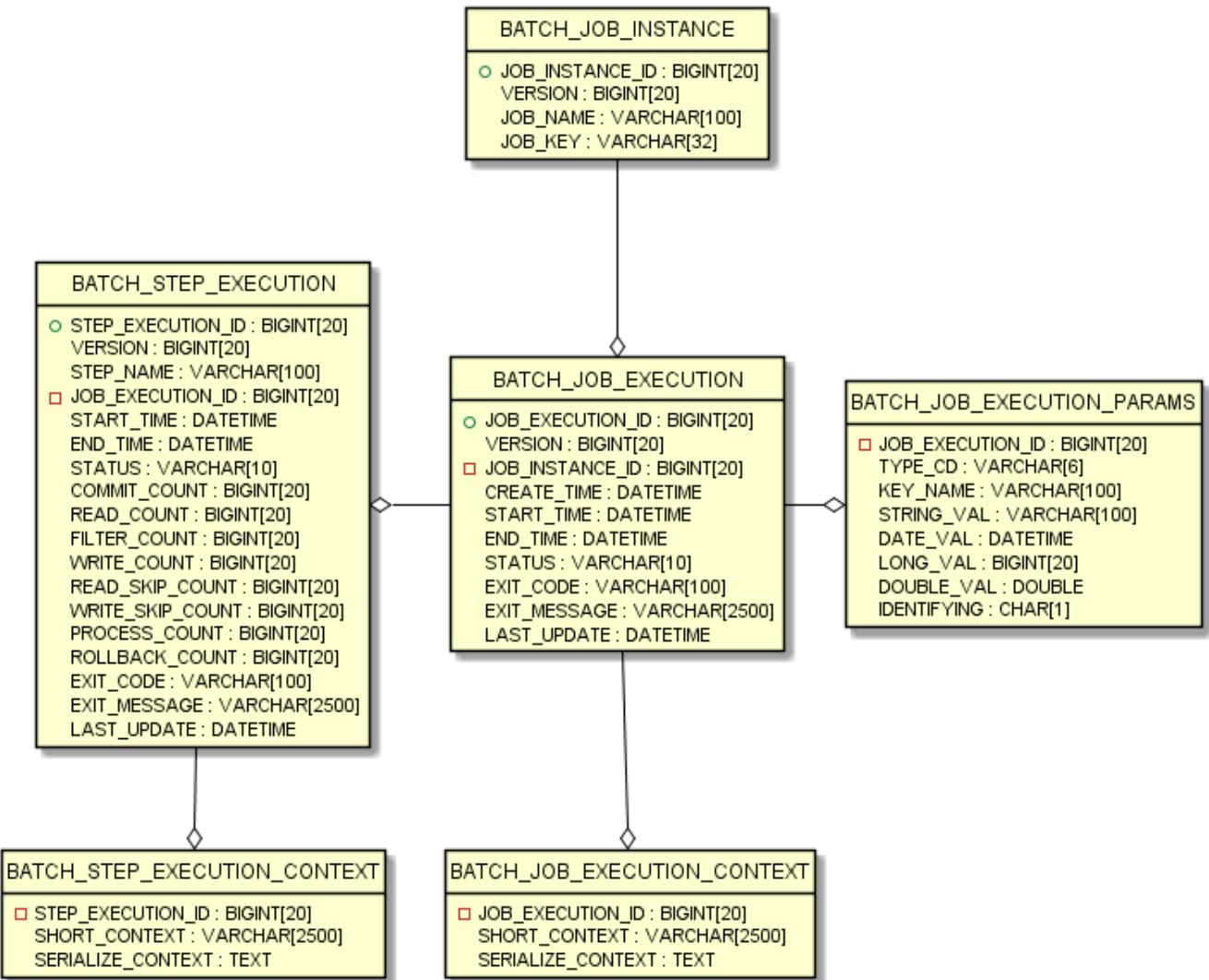
Spring Batchメタデータテーブルは、Javaでそれらを表すドメインオブジェクト(Entityオブジェクト)に対応している。

#### 対応一覧

テーブル	Entityオブジェクト	概要
BATCH_JOB_INSTANCE	JobInstance	ジョブ名、およびジョブパラメータをシリアル化した文字列を保持する。
BATCH_JOB_EXECUTION	JobExecution	ジョブの状態・実行結果を保持する。
BATCH_JOB_EXECUTION_PARAMS	JobExecutionParams	起動時に与えられたジョブパラメータを保持する。
BATCH_JOB_EXECUTION_CONTEXT	JobExecutionContext	ジョブ内部のコンテキストを保持する。
BATCH_STEP_EXECUTION	StepExecution	ステップの状態・実行結果、コミット・ロールバック件数を保持する。
BATCH_STEP_EXECUTION_CONTEXT	StepExecutionContext	ステップ内部のコンテキストを保持する。

JobRepositoryは、各Javaオブジェクトに保存された内容を、テーブルへ正確に格納する責任がある。

6つの全テーブルと相互関係のERDモデルはを以下に示す。



ER図

#### 2.3.2.4.1. バージョン

データベーステーブルの多くは、バージョンカラムが含まれてる。Spring Batchは、データベースへの更新を扱う楽観的ロック戦略を採用しているため、このカラムは重要となる。このレコードは、バージョンカラムの値がインクリメントされるたびに更新されることを意味している。JobRepositoryが値の更新時に、バージョン番号が変更されている場合、同時アクセスのエラーが発生したことを示すOptimisticLockingFailureExceptionがスローされる。別のバッチジョブは異なるマシンで実行されているかもしれないが、それらはすべて同じデータベーステーブルを使用しているため、このチェックが必要となる。

#### 2.3.2.4.2. ID(シーケンス)定義

BATCH\_JOB\_INSTANCE、BATCH\_JOB\_EXECUTION、およびBATCH\_STEP\_EXECUTIONは各\_IDで終わる列が含まれている。これらのフィールドは、それぞれのテーブル用主キーとして機能する。しかし、それらは、データベースで生成されたキーではなく、むしろ個別のシーケンスで生成される。データベースにドメインオブジェクトのいずれかを挿入した後、それが与えられたキーは、それらが一意にJavaで識別できるように、実際のオブジェクトに設定する必要なためである。

データベースによってはシーケンスをサポートしていないことがある。この場合、テーブルを各シーケンスの代わりに使用している。

#### 2.3.2.4.3. テーブル定義

各テーブルの項目について説明をする。

#### BATCH\_JOB\_INSTANCE

BATCH\_JOB\_INSTANCEテーブルはJobInstanceに関連するすべての情報を保持し、全体的な階層の最上位である。

##### BATCH\_JOB\_INSTANCEの定義

カラム名	説明
JOB_INSTANCE_ID	インスタンスを識別する一意のIDで主キーである。
VERSION	<a href="#">バージョンを参照。</a>
JOB_NAME	ジョブの名前。 インスタンスを識別するために必要とされるので非nullである。
JOB_KEY	同じジョブを別々のインスタンスとして一意に識別するためのシリアル化されたJobParameters。 同じジョブ名をもつJobInstancesは、異なるJobParameters(つまり、異なるJOB_KEY値)をもつ必要がある。

#### BATCH\_JOB\_EXECUTION

BATCH\_JOB\_EXECUTIONテーブルはJobExecutionオブジェクトに関連するすべての情報を保持する。 ジョブが実行されるたびに、常に新しいJobExecutionでこの表に新しい行が登録される。

##### BATCH\_JOB\_EXECUTIONの定義

カラム名	説明
JOB_EXECUTION_ID	一意にこのジョブ実行を識別する主キー。
VERSION	<a href="#">バージョンを参照。</a>
JOB_INSTANCE_ID	このジョブ実行が属するインスタンスを示すBATCH_JOB_INSTANCEテーブルからの外部キー。 インスタンスごとに複数の実行が存在する場合がある。
CREATE_TIME	ジョブ実行が作成された時刻。
START_TIME	ジョブ実行が開始された時刻。
END_TIME	ジョブ実行が成功または失敗に関係なく、終了した時刻を表す。 ジョブが現在実行されていないにもかかわらず、このカラムの値が空であることは、いくつかのエラータイプがあり、フレームワークが最後のセーブを実行できなかったことを示す。
STATUS	ジョブ実行のステータスを表す文字列。BatchStatus列挙オブジェクトが 出力する文字列である。
EXIT_CODE	ジョブ実行の終了コードを表す文字列。 CommandLineJobRunnerによる起動の場合、これを数値に変換する ことができる。

カラム名	説明
EXIT_MESSAGE	ジョブが終了状態のより詳細な説明を表す文字列。 障害が発生した場合には、可能であればスタックトレースができるだけ多く含む文字列となる場合がある。
LAST_UPDATED	このレコードのジョブ実行が最後に更新された時刻。

## BATCH\_JOB\_EXECUTION\_PARAMS

BATCH\_JOB\_EXECUTION\_PARAMSテーブルは、JobParametersオブジェクトに関連するすべての情報を保持する。これはジョブに渡された0以上のキーと値とのペアが含まれ、ジョブが実行されたパラメータを記録する役割を果たす。

### BATCH\_JOB\_EXECUTION\_PARAMSの定義

カラム名	説明
JOB_EXECUTION_ID	このジョブパラメータが属するジョブ実行を示すBATCH_JOB_EXECUTIONテーブルからの外部キー。
TYPE_CD	String、date、long、またはdoubleのいずれかのデータ型であることを示す文字列。
KEY_NAME	パラメータキー。
STRING_VAL	データ型が文字列である場合のパラメータ値。
DATE_VAL	データ型が日時である場合のパラメータ値。
LONG_VAL	データ型が整数値である場合のパラメータ値。
DOUBLE_VAL	データ型が実数である場合のパラメータ値。
IDENTIFYING	パラメータがジョブインスタンスが一意であることを識別するための値であることを示すフラグ。

#### ジョブパラメータの制約について



- BATCH\_JOB\_EXECUTION\_PARAMSに格納するため、パラメータが取りうる値にはサイズによる制限がある。
- マルチバイト文字を使用する場合は、使用するエンコーディングによりSTRING\_VALのサイズを調整することを検討する。

## BATCH\_JOB\_EXECUTION\_CONTEXT

BATCH\_JOB\_EXECUTION\_CONTEXTテーブルは、JobのExecutionContextに関連するすべての情報は保持する。特定のジョブ実行に必要とされるジョブレベルのデータがすべて含まれている。このデータは、ジョブが失敗した後で処理を再処理する際に取得しなければならない状態を表し、失敗したジョブが「処理を中断したところから始める」ことを可能にする。

### BATCH\_JOB\_EXECUTION\_CONTEXTの定義

カラム名	説明
JOB_EXECUTION_ID	このJobのExecutionContextが属するジョブ実行を示すBATCH_JOB_EXECUTIONテーブルからの外部キー。

カラム名	説明
SHORT_CONTEXT	SERIALIZED_CONTEXTの文字列表現。
SERIALIZED_CONTEXT	シリアル化されたコンテキスト全体。

## BATCH\_STEP\_EXECUTION

BATCH\_STEP\_EXECUTIONテーブルは、StepExecutionオブジェクトに関連するすべての情報を保持する。このテーブルには、BATCH\_JOB\_EXECUTIONテーブルと多くの点で非常に類似しており、各JobExecutionが作られるごとに常にStepごとに少なくとも1つのエントリがある。

### BATCH\_STEP\_EXECUTIONの定義

カラム名	説明
STEP_EXECUTION_ID	一意にこのステップ実行を識別する主キー。
VERSION	<a href="#">バージョン</a> を参照。
STEP_NAME	ステップの名前。
JOB_EXECUTION_ID	このStepExecutionが属するJobExecutionを示すBATCH_JOB_EXECUTIONテーブルからの外部キー。
START_TIME	ステップ実行が開始された時刻。
END_TIME	ステップ実行が成功または失敗に関係なく、終了した時刻を表す。 ジョブが現在実行されていないにもかかわらず、このカラムの値が空であることは、いくつかのエラータイプがあり、フレームワークが最後のセーブを実行できなかったことを示す。
STATUS	ステップ実行のステータスを表す文字列。BatchStatus列挙オブジェクトが出力する文字列である。
COMMIT_COUNT	トランザクションをコミットしている回数。
READ_COUNT	ItemReaderで読み込んだデータ件数。
FILTER_COUNT	ItemProcessorでフィルタリングしたデータ件数。
WRITE_COUNT	ItemWriterで書き込んだデータ件数。
READ_SKIP_COUNT	ItemReaderでスキップしたデータ件数。
WRITE_SKIP_COUNT	ItemWriterでスキップしたデータ件数。
PROCESS_SKIP_COUNT	ItemProcessorでスキップしたデータ件数。
ROLLBACK_COUNT	トランザクションをロールバックしている回数。
EXIT_CODE	ステップ実行の終了コードを表す文字列。 CommandLineJobRunnerによる起動の場合、これを数値に変換することができる。
EXIT_MESSAGE	ステップが終了状態のより詳細な説明を表す文字列。 障害が発生した場合には、可能であればスタックトレースをできるだけ多く含む文字列となる場合がある。
LAST_UPDATED	このレコードのステップ実行が最後に更新された時刻。

## BATCH\_STEP\_EXECUTION\_CONTEXT

BATCH\_STEP\_EXECUTION\_CONTEXTテーブルは、StepのExecutionContextに関連するすべての情報を保持する。特定のステップ実行に必要とされるステップレベルのデータがすべて含まれている。このデータは、ジョブが失敗した後で処理を再処理する際に取得しなければならない状態を表し、失敗したジョブが「処理を中断したところから始める」ことを可能にする。

### BATCH\_STEP\_EXECUTION\_CONTEXTの定義

カラム名	説明
STEP_EXECUTION_ID	このStepのExecutionContextが属するジョブ実行を示すBATCH_STEP_EXECUTIONテーブルからの外部キー。
SHORT_CONTEXT	SERIALIZED_CONTEXTの文字列表現。
SERIALIZED_CONTEXT	シリアル化されたコンテキスト全体。

#### 2.3.2.4.4. DDLスクリプト

Spring Batch CoreのJARファイルには、いくつかのデータベースプラットフォームに応じたリレーショナル表を作成するサンプルスクリプトが含まれている。これらのスクリプトはそのまま使用、または必要に応じて追加のインデックスと制約を変更することができる。

スクリプトは、org.springframework.batch.coreのパッケージに含まれており、ファイル名は、`schema-*.sql`で形成されている。`"*"`は、ターゲット・データベース・プラットフォームの短い名前である。

#### 2.3.2.5. 代表的な性能チューニングポイント

Spring Batchにおける代表的な性能チューニングポイントを説明する。

##### チャンクサイズの調整

リソースへの出力によるオーバヘッドを抑えるために、チャンクサイズを大きくする。

ただし、チャンクサイズを大きくしすぎるとリソース側の負荷が高くなりかえって性能が低下があるので、適度なサイズになるように調整を行う。

##### フェッチサイズの調整

リソースからの入力によるオーバヘッドを抑えるために、リソースに対するフェッチサイズ(バッファサイズ)を大きくする。

##### ファイル読み込みの効率化

BeanWrapperFieldSetMapperを使用すると、Beanのクラスとプロパティ名を順番に指定するだけでレコードをBeanにマッピングしてくれる。しかし、内部で複雑な処理を行うため時間がかかる。マッピングを行う専用のFieldSetMapperインターフェース実装を用いることで処理時間を短縮できる可能性がある。

ファイル入出力の詳細は、[ファイルアクセス](#)を参照。

##### 並列処理・多重処理

Spring Batchでは、Step実行の並列化、データ分割による多重処理をサポートしている。並列化もしくは多重化を行い、処理を並列走行させることで性能を改善できる。しかし、並列数および多重数を大きくしすぎるとリソース側の負荷が高くなりかえって性能が低下があるので、適度なサイズになるように調整を行う。

並列処理・多重処理の詳細は、[並列処理と多重処理](#)を参照。

## 分散処理の検討

Spring Batchでは、複数マシンでの分散処理もサポートしている。指針は、並列処理・多重処理と同様である。

分散処理は、基盤設計や運用設計が複雑化するため、本ガイドラインでは説明を行わない。

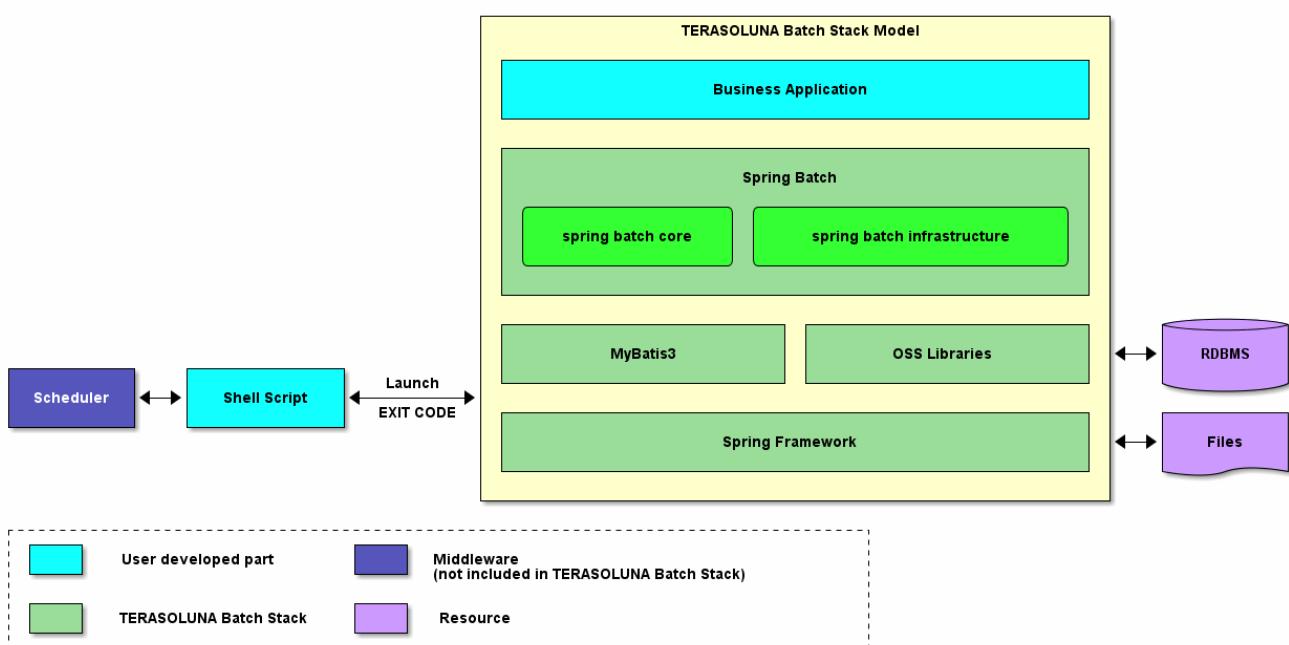
## 2.4. TERASOLUNA Batch Framework for Java (5.x)のアーキテクチャ

### 2.4.1. 概要

TERASOLUNA Batch Framework for Java (5.x)のアーキテクチャ全体像を説明する。

TERASOLUNA Batch Framework for Java (5.x)では、[一般的なバッチ処理システム](#)で説明したとおりSpring Batchを中心としたOSSの組み合わせを利用して実現する。

Spring Batchの階層アーキテクチャを含めたTERASOLUNA Batch Framework for Java (5.x)の構成概略図を以下に示す。



TERASOLUNA Batch Framework for Java (5.x)の構成概略図

Spring Batchの階層アーキテクチャの説明

アプリケーション

開発者によって書かれたすべてのジョブ定義およびビジネスロジック。

コア

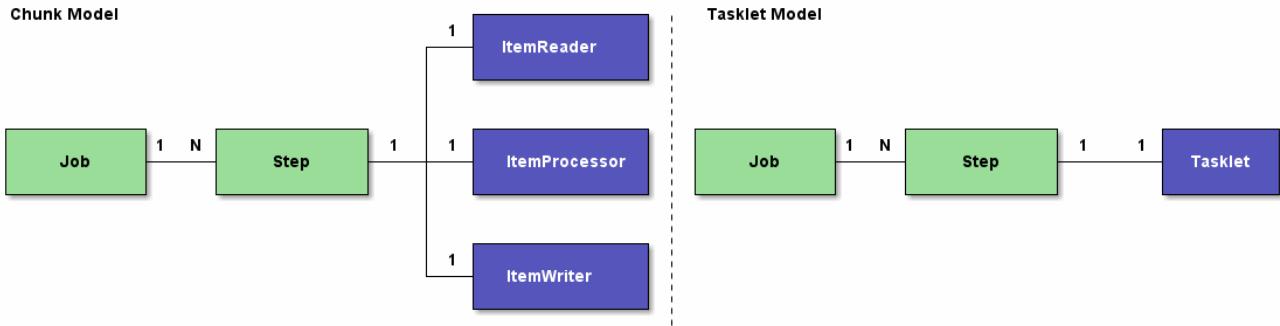
Spring Batch が提供するバッチジョブを起動し、制御するために必要なコア・ランタイム・クラス。

インフラストラクチャ

Spring Batch が提供する開発者およびコアフレームワーク自体が利用する一般的なItemReader/ItemProcessor/ItemWriterの実装。

### 2.4.2. ジョブの構成要素

ジョブの構成要素を説明するため、ジョブの構成概略図を下記に示す。



## ジョブの構成概略図

この節では、ジョブとステップについて構成すべき粒度の指針も含めて説明をする。

### 2.4.2.1. ジョブ

ジョブとは、バッチ処理全体をカプセル化するエンティティであり、ステップを格納するためのコンテナである。

1つのジョブは、1つ以上のステップで構成することができる。

ジョブの定義は、XMLによるBean定義ファイルに記述する。ジョブ定義ファイルには複数のジョブを定義することができるが、ジョブの管理が煩雑になりやすくなる。

従って、TERASOLUNA Batch Framework for Java (5.x)では以下の指針とする。

☞ 1ジョブ=1ジョブ定義ファイル

### 2.4.2.2. ステップ

ステップとは、バッチ処理を制御るために必要な情報を定義したものである。ステップにはチャンクモデルとタスクレットモデルを定義することができる。

#### チャンクモデル

- ItemReader、ItemProcessor、およびItemWriterで構成される。

#### タスクレットモデル

- Taskletだけで構成される。

[バッチ処理で考慮する原則と注意点](#)にあるとおり、単一のバッチ処理では、可能な限り簡素化し、複雑な論理構造を避ける必要がある。

従って、TERASOLUNA Batch Framework for Java (5.x)では以下の指針とする。

☞ 1ステップ=1バッチ処理=1ビジネスロジック



### チャンクモデルでのビジネスロジック分割

1つのビジネスロジックが複雑で規模が大きくなる場合、ビジネスロジックを分割することがある。概略図を見るとわかるとおり、1つのステップには1つのItemProcessorしか設定できないため、ビジネスロジックの分割ができないように思える。しかし、CompositeItemProcessorという複数のItemProcessorをまとめるItemProcessorがあり、この実装を使うことでビジネスロジックを分割して実行することができる。

## 2.4.3. ステップの実装方式

### 2.4.3.1. チャンクモデル

チャンクモデルの定義と使用目的を説明する。

#### 定義

ItemReader、ItemProcessorおよびItemWriter実装とチャンク数をChunkOrientedTaskletに設定する。それぞれの役割を説明する。

- ChunkOrientedTasklet…ItemReader/ItemProcessorを呼び出し、チャンクを作成する。作成したチャンクをItemWriterへ渡す。
- ItemReader…入力データを読み込む。
- ItemProcessor…読み込んだデータを加工する。
- ItemWriter…加工されたデータをチャンク単位で出力する。

チャンクモデルの概要は、[チャンクモデル](#)を参照。

#### チャンクモデルのジョブ設定例

```
<batch:job id="exampleJob">
    <batch:step id="exampleStep">
        <batch:tasklet>
            <batch:chunk reader="reader"
                          processor="processor"
                          writer="writer"
                          commit-interval="100" />
        </batch:tasklet>
    </batch:step>
</batch:job>
```

#### 使用目的

一定件数のデータをまとめて処理を行うため、大量データを取り扱う場合に用いられる。

### 2.4.3.2. タスクレットモデル

タスクレットモデルの定義・使用目的を説明する。

#### 定義

Tasklet実装だけを設定する。

タスクレットモデルの概要は、[タスクレットモデル](#)を参照。

#### . タスクレットモデルのジョブ設定例

```
----  
<batch:job id="exampleJob">  
  <batch:step id="exampleStep">  
    <batch:tasklet ref="myTasklet">  
    </batch:tasklet>  
  </batch:step>  
</batch:job>  
----
```

#### 使用目的

システムコマンドの実行など、入出力を伴わない処理を実行するために用いられる。  
また、一括でデータをコミットしたい場合にも用いられる。

#### 2.4.3.3. チャンクモデルとタスクレットモデルの機能差

チャンクモデルとタスクレットモデルの機能差について説明する。ここでは、詳細については各機能の節を参照してもらい、ここでは概略のみにとどめる。

#### 機能差一覧

機能	チャンクモデル	タスクレットモデル
構成要素	ItemReader/ItemProcessor/ItemWriter /ChunkOrientedTaskletで構成される。	Taksletのみで構成される。
トランザクション	チャンク単位にトランザクションが発生する。	1トランザクションで処理する。
推奨する再処理方式	リラン、リストートを利用できる。	リランのみ利用することを原則とする。
例外ハンドリング	リスナーを使うことでハンドリング処理が容易になっている。独自実装も可能である。	独自実装が必要である。

#### 2.4.4. ジョブの起動方式

ジョブの起動方式について説明する。ジョブの起動方式には以下のものがある。

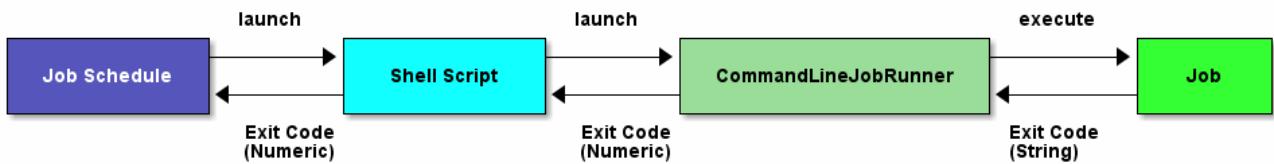
- 同期実行方式
- 非同期実行方式

それぞれの起動方式について説明する。

##### 2.4.4.1. 同期実行方式

同期実行方式とは、ジョブを起動してからジョブが終了するまで起動元へ制御が戻らない実行方式である。

ジョブスケジューラからジョブを起動する概略図を示す。



1. ジョブスケジューラからジョブを起動するためのシェルスクリプトを起動する。  
シェルスクリプトから終了コード(数値)が返却するまでジョブスケジューラは待機する。
2. シェルスクリプトからジョブを起動するために**CommandLineJobRunner**を起動する。  
**CommandLineJobRunner**から終了コード(数値)が返却するまでシェルスクリプトは待機する。
3. **CommandLineJobRunner**はジョブを起動する。ジョブは処理終了後に終了コード(文字列)を**CommandLineJobRunner**へ返却する。  
**CommandLineJobRunner**は、ジョブから返却された終了コード(文字列)から終了コード(数値)に変換してシェルスクリプトへ返却する。

#### 2.4.4.2. 非同期実行方式

非同期実行方式とは、起動元とは別の実行基盤(別スレッドなど)でジョブを実行することで、ジョブ起動後すぐに起動元へ制御が戻る方式である。この方式の場合、ジョブの実行結果はジョブ起動とは別の手段で取得する必要がある。

TERASOLUNA Batch Framework for Java (5.x)では、以下に示す2とおりの方法について説明をする。

- 非同期実行方式(DBポーリング)
- 非同期実行方式(Webコンテナ)

##### その他の非同期実行方式

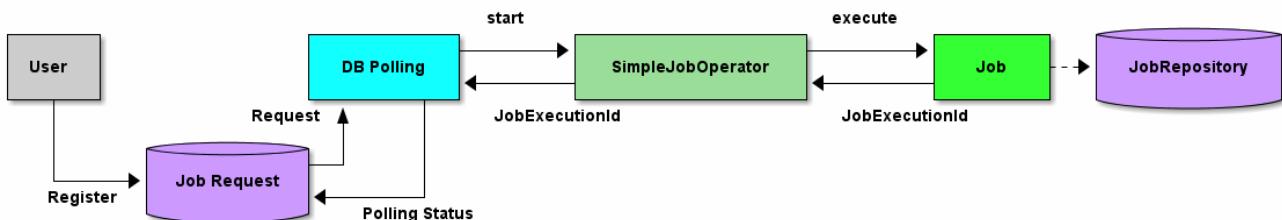


MQなどのメッセージを利用して非同期実行を実現することもできるが、ジョブ実行のポイントは同じであるため、{batch5\_guide}では説明は割愛する。

##### 2.4.4.2.1. 非同期実行方式(DBポーリング)

非同期実行(DBポーリング)とは、ジョブ実行の要求をデータベースに登録し、その要求をポーリングして、ジョブを実行する方式である。

TERASOLUNA Batch Framework for Java (5.x)は、DBポーリング機能を提供している。提供しているDBポーリングによる起動の概略図を示す。



1. ユーザはデータベースへジョブ要求を登録する。
2. DBポーリング機能は、定期的にジョブ要求の登録を監視していて、登録されたことを検知すると該当するジョブを実行する。
  - SimpleJobOperatorからジョブを起動し、ジョブ終了後にJobExecutionIdを受け取る。
  - JobExecutionIdとは、ジョブ実行を一意に識別するIDであり、このIDを使ってJobRepositoryから実行結果を参照する。
  - ジョブの実行結果は、Spring Batchの仕組みによって、JobRepositoryへ登録される。
  - DBポーリング自体が非同期で実行されている。
3. DBポーリング機能は、SimpleJobOperatorから返却されたJobExecutionIdとスタートスを起動したジョブ要求に対して更新を行う。
4. ジョブの処理経過・結果は、JobExecutionIdを利用して別途参照を行う。

#### 2.4.4.2.2. 非同期実行方式(Webコンテナ)

**非同期実行(Webコンテナ)**とは、Webコンテナ上のWebアプリケーションへのリクエストを契機にジョブを非同期実行する方式である。Webアプリケーションは、ジョブの終了を待たずに起動後すぐにレスポンスを返却することができる。



Webコンテナ概略図

1. クライアントからWebアプリケーションへリクエストを送信する。
2. Webアプリケーションは、リクエストから要求されたジョブを非同期実行する。
  - SimpleJobOperatorからジョブを起動直後にJobExecutionIdを受け取る。
  - ジョブの実行結果は、Spring Batchの仕組みによって、JobRepositoryへ登録される。
3. Webアプリケーションは、ジョブの終了を待たずにクライアントへレスポンスを返信する。
4. ジョブの処理経過・結果は、JobExecutionIdを利用して別途参照を行う。

また、[TERASOLUNA Server Framework for Java \(5.x\)](#)で構築されるWebアプリケーションと連携することも可能である。

#### 2.4.5. 利用する際の検討ポイント

TERASOLUNA Batch Framework for Java (5.x)を利用する際の検討ポイントを示す。

##### ジョブ起動方法

###### 同期実行方式

スケジュールどおりにジョブを起動したり、複数のジョブを組み合わせてバッチ処理行う場合に利用する。

## 非同期実行方式(DBポーリング)

ディレード処理、処理時間が短いジョブの連続実行、大量ジョブの集約などに利用する。

## 非同期実行方式(Webコンテナ)

DBポーリングと同様だが、起動までの即時性が求められる場合にはこちらを利用する。

### 実装方式

#### チャンクモデル

大量データを効率よく処理したい場合に利用する。

#### タスクレットモデル

シンプルな処理や、定型化しにくい処理、データを一括で処理したい場合に利用する。

# Chapter 3. アプリケーション開発の流れ

## 3.1. バッチアプリケーションの開発

バッチアプリケーションの開発について、以下の流れで説明する。

- [ランクプロジェクトとは](#)
- [プロジェクトの作成](#)
- [プロジェクトの構成](#)
- [開発の流れ](#)
- [アプリケーションのビルド](#)

### 3.1.1. ブランクプロジェクトとは

ブランクプロジェクトとは、Spring BatchやMyBatis3をはじめとする各種設定を予め行った開発プロジェクトの雛形であり、アプリケーション開発のスタート地点である。

本ガイドラインでは、シングルプロジェクト構成のブランクプロジェクトを提供する。

構成の説明については、[プロジェクトの構成](#)を参照のこと。

#### TERASOLUNA Server 5.xとの違い

TERASOLUNA Server 5.xはマルチプロジェクト構成を推奨している。この理由は主に、以下の様なメリットを享受するためである。

- 環境差分の吸収しやすくする
- ビジネスロジックとプレゼンテーションを分離しやすくする



しかし、本ガイドラインではTERASOLUNA Server 5.xと異なりシングルプロジェクト構成としている。

これは、前述の点はバッチアプリケーションの場合においても考慮すべきだが、シングルプロジェクト構成にすることで1ジョブに関連する資材を近づけることを優先している。

また、バッチアプリケーションの場合、環境差分はプロパティファイルや環境変数で切替れば十分なケースが多いことも理由の1つである。

### 3.1.2. プロジェクトの作成

Maven Archetype Pluginの[archetype:generate](#)を使用して、プロジェクトを作成する方法を説明する。

作成環境の前提について  
以下を前提とし説明する。

- Java SE Development Kit 8

- Apache Maven 3.x



- インターネットに繋がっていること
- インターネットにプロキシ経由で繋ぐ場合は、Mavenのプロキシ設定が行われていること
- IDE
  - Spring Tool Suite / Eclipse 等

プロジェクトを作成するディレクトリにて、以下のコマンドを実行する。

コマンドプロンプト(*Windows*)

```
C:\xxx> mvn archetype:generate^
-DarchetypeGroupId=org.terasoluna.batch^
-DarchetypeArtifactId=terasoluna-batch-archetype^
-DarchetypeVersion=5.0.0.RELEASE
```

Bash(*Unix, Linux, ...*)

```
$ mvn archetype:generate \
-DarchetypeGroupId=org.terasoluna.batch \
-DarchetypeArtifactId=terasoluna-batch-archetype \
-DarchetypeVersion=5.0.0.RELEASE
```

その後、利用者の状況に合わせて、以下を対話式に設定する。

- groupId
- artifactId
- version
- package

以下の値を設定し実行した例を示す。

プランクプロジェクトの各要素の説明

項目名	設定例
groupId	com.example.batch
artifactId	batch
version	1.0.0-SNAPSHOT
package	com.example.batch

## 実行例

```
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----
[INFO]
[INFO] >>> maven-archetype-plugin:2.4:generate (default-cli) > generate-sources @
standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.4:generate (default-cli) < generate-sources @
standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.4:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Interactive mode
```

(.. omitted)

```
Define value for property 'groupId': : com.example.batch
Define value for property 'artifactId': : batch
Define value for property 'version': 1.0-SNAPSHOT: : 1.0.0-SNAPSHOT
Define value for property 'package': com.example.batch: :
Confirm properties configuration:
groupId: com.example.batch
artifactId: batch
version: 1.0.0-SNAPSHOT
package: com.example.batch
Y: : y
[INFO] -----
[INFO] Using following parameters for creating project from Archetype: terasoluna-
batch-archetype:5.0.0-SNAPSHOT
[INFO] -----
[INFO] Parameter: groupId, Value: com.example.batch
[INFO] Parameter: artifactId, Value: batch
[INFO] Parameter: version, Value: 1.0.0-SNAPSHOT
[INFO] Parameter: package, Value: com.example.batch
[INFO] Parameter: packageInPathFormat, Value: com/example/batch
[INFO] Parameter: package, Value: com.example.batch
[INFO] Parameter: version, Value: 1.0.0-SNAPSHOT
[INFO] Parameter: groupId, Value: com.example.batch
[INFO] Parameter: artifactId, Value: batch
[INFO] project created from Archetype in dir: C:\workspaces\zzz\batch
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 02:56 min
[INFO] Finished at: 2017-02-07T17:09:52+09:00
[INFO] Final Memory: 16M/240M
[INFO] -----
```

以上により、プロジェクトの作成が完了した。

正しく作成出来たかどうかは、以下の要領で確認できる。

正しく作成できたことの確認(*Bash*)

```
$ mvn clean dependency:copy-dependencies -DoutputDirectory=lib package  
$ java -cp 'lib/*:target/*'  
org.springframework.batch.core.launch.support.CommandLineJobRunner \  
META-INF/jobs/job01/job01.xml job01
```

以下の出力が得られれば正しく作成できている。

## 出力例

```
$ mvn clean dependency:copy-dependencies -DoutputDirectory=lib package
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building TERASOLUNA Batch Framework for Java (5.x) Blank Project 1.0.0-SNAPSHOT
[INFO] -----
[INFO]

(.. omitted)

[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3.618 s
[INFO] Finished at: 2017-02-07T17:32:27+09:00
[INFO] Final Memory: 26M/250M
[INFO] -----
```

```
$ java -cp 'lib/*;target/*'
org.springframework.batch.core.launch.support.CommandLineJobRunner META-
INF/jobs/job01/job01.xml job01
[2017/02/07 17:35:26] [main] [o.s.c.s.ClassPathXmlApplicationContext] [INFO ] Refreshing
org.springframework.context.support.ClassPathXmlApplicationContext@62043840: startup
date [Tue Feb 07 17:35:26 JST 2017]; root of context hierarchy
(.. ommited)
[2017/02/07 17:35:27] [main] [o.s.b.c.l.s.SimpleJobLauncher] [INFO ] Job: [FlowJob:
[name=job01]] launched with the following parameters: [{jsr_batch_run_id=1}]
[2017/02/07 17:35:27] [main] [o.s.b.c.j.SimpleStepHandler] [INFO ] Executing step:
[job01.step01]
[2017/02/07 17:35:27] [main] [o.s.b.c.l.s.SimpleJobLauncher] [INFO ] Job: [FlowJob:
[name=job01]] completed with the following parameters: [{jsr_batch_run_id=1}] and the
following status: [COMPLETED]
[2017/02/07 17:35:27] [main] [o.s.c.s.ClassPathXmlApplicationContext] [INFO ] Closing
org.springframework.context.support.ClassPathXmlApplicationContext@62043840: startup
date [Tue Feb 07 17:35:26 JST 2017]; root of context hierarchy
```

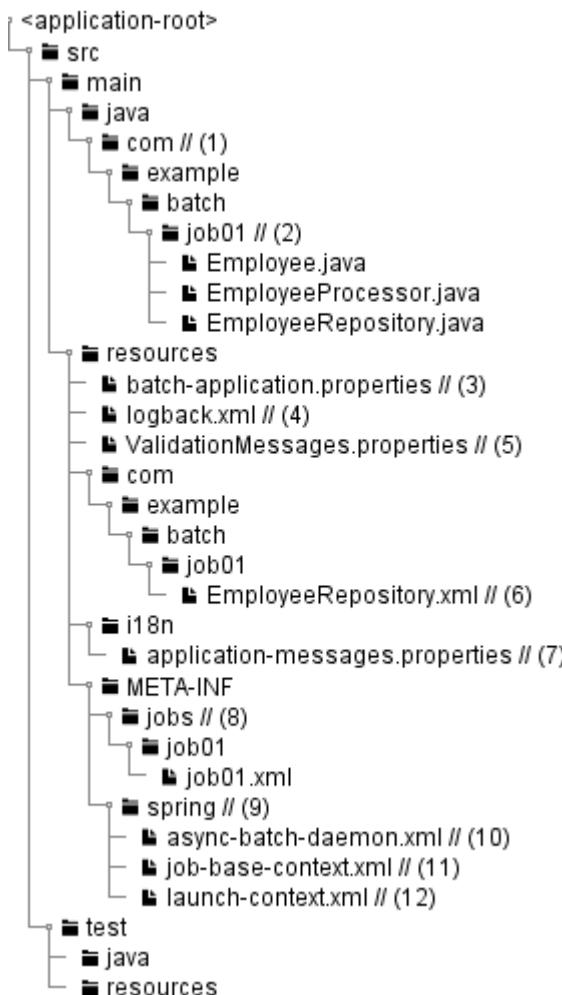
### 3.1.3. プロジェクトの構成

前述までで作成したプロジェクトの構成について説明する。プロジェクトは、以下の点を考慮した構成となっている。

- 起動方式に依存しないジョブの実装を実現する
- Spring BatchやMyBatisといった各種設定の手間を省く
- 環境依存の切替を容易にする

以下に構成を示し、各要素について説明する。

(わかりやすさのため、前述のmvn archetype:generate実行時の出力を元に説明する。)



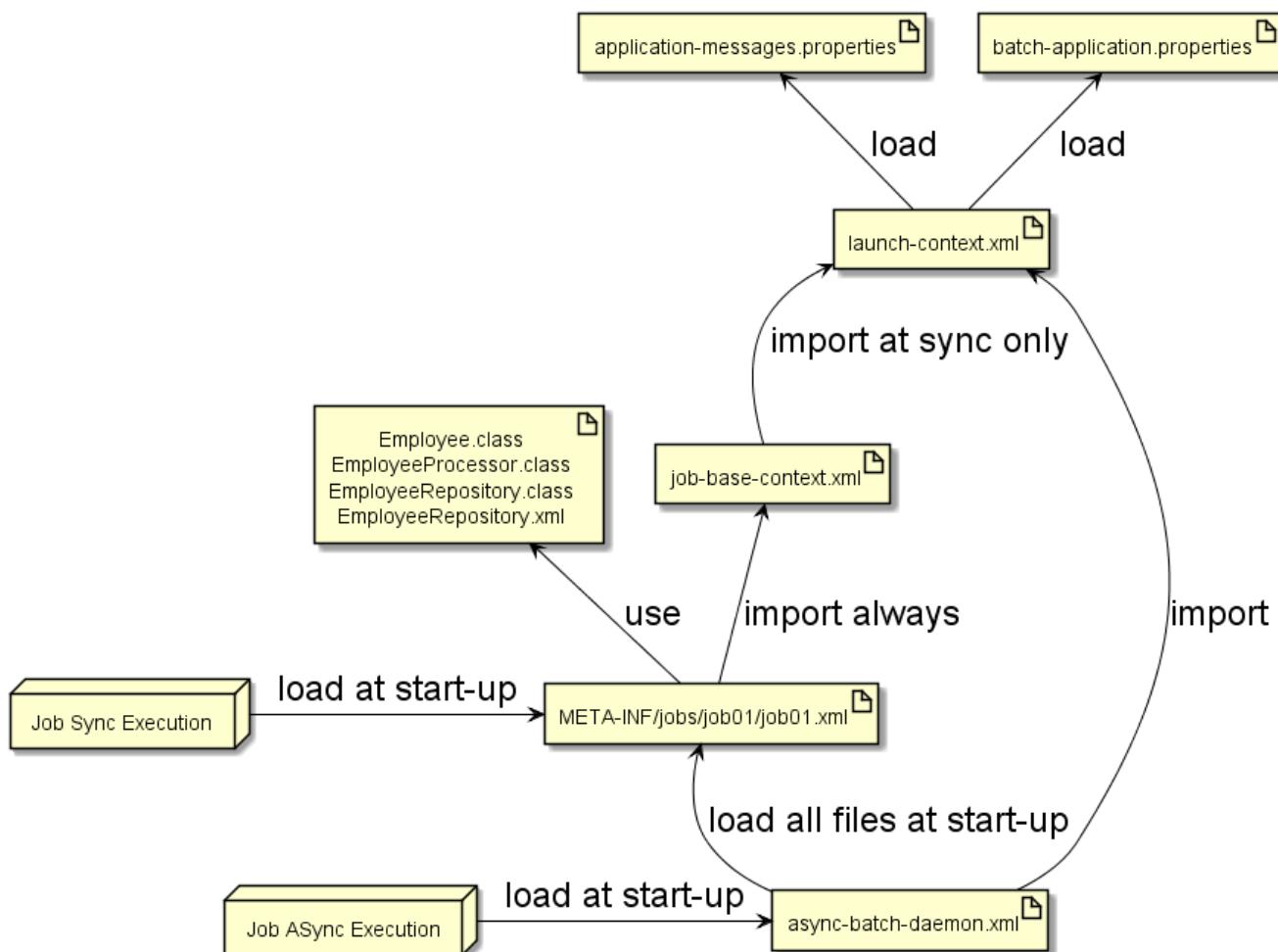
## プロジェクトのディレクトリ構造

### プランクプロジェクトの各要素の説明

項番	説明
(1)	バッチアプリケーション全体の各種クラスを格納するrootパッケージ。
(2)	1ジョブに関する各種クラスを格納するパッケージ。 ここには、DTO、TaskletやProcessorの実装、MyBatis3のMapperインターフェースを格納する。 本ガイドラインでは格納方法に制約は設けないので、これは一例として参考にしてほしい。 初期状態を参考にユーザにて自由にカスタムしてよいが、 ジョブ固有の資材を判断しやすくすることに配慮してほしい。
(3)	バッチアプリケーション全体に関わる設定ファイル。 初期状態では、データベースの接続や、非同期実行に関する設定を記述している。 ユーザにて、自由に追記してよい。
(4)	Logback(ログ出力)の設定ファイル。
(5)	BeanValidationを用いた入力チェックにて、エラーとなった際に表示するメッセージを定義する設定ファイル。 初期状態では、BeanValidationと、その実装であるHibernateValidatorのデフォルトメッセージを定義したうえで、すべてコメントアウトしている。 この状態ではデフォルトメッセージを使うため、メッセージをカスタマイズしたい場合にのみコメントインし任意のメッセージに修正すること。

項目番	説明
(6)	MyBatis3のMapperインターフェースの対となるMapper XMLファイル。
(7)	主にログ出力時に用いるメッセージを定義するプロパティファイル。
(8)	ジョブ固有のBean定義ファイルを格納するディレクトリ。 階層構造はジョブ数に応じて自由に構成してよい。
(9)	バッチアプリケーション全体に関わるBean定義ファイルを格納するディレクトリ。 Spring BatchやMyBatisの初期設定や、同期 /非同期といった起動契機に依らずにジョブを起動するための設定を行っている。
(10)	非同期実行(DBポーリング)機能に関連する設定を記述したBean定義ファイル。
(11)	ジョブ固有のBean定義ファイルにてimportすることで、各種設定を削減するためのBean定義 ファイル。 これをimportすることで、ジョブは起動契機によるBean定義の差を吸収することが出来る。
(12)	Spring Batchの挙動や、ジョブ共通の設定に対するBean定義ファイル。

また、各ファイルの関連図を以下に示す。



各ファイルの関連図

### 3.1.4. 開発の流れ

ジョブを開発する一連の流れについて説明する。

ここでは、詳細な説明ではなく、大まかな流れを把握することを主眼とする。

### 3.1.4.1. IDEへの取り込み

生成したプロジェクトはMavenのプロジェクト構成に従っているため、各種IDEによって、Mavenプロジェクトとしてimportする。

詳細な手順は割愛する。

### 3.1.4.2. アプリケーション全体の設定

ユーザの状況に応じて以下をカスタマイズする。

- pom.xmlのプロジェクト情報
- データベース関連の設定

これら以外の設定をカスタマイズする方法については、個々の機能にて説明する。

#### 3.1.4.2.1. pom.xmlのプロジェクト情報

プロジェクトのPOMには以下の情報が仮の値で設定されているため、状況に応じて設定すること。

- プロジェクト名(name要素)
- プロジェクト説明(description要素)
- プロジェクトURL(url要素)
- プロジェクト創設年(inceptionYear要素)
- プロジェクトライセンス(licenses要素)
- プロジェクト組織(organization要素)

#### 3.1.4.2.2. データベース関連の設定

データベース関連の設定は複数箇所にあるため、それぞれを修正すること。

pom.xml

```
<!-- (1) -->
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>

<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
```

*batch-application.properties*

```
# (2)
# Admin DataSource settings.
admin.jdbc.driver=org.h2.Driver
admin.jdbc.url=jdbc:h2:mem:batch-admin;DB_CLOSE_DELAY=-1
admin.jdbc.username=sa
admin.jdbc.password=

# (2)
# Job DataSource settings.
#jdbc.driver=org.postgresql.Driver
#jdbc.url=jdbc:postgresql://localhost:5432/postgres
#jdbc.username=postgres
#jdbc.password=postgres
jdbc.driver=org.h2.Driver
jdbc.url=jdbc:h2:mem:batch;DB_CLOSE_DELAY=-1
jdbc.username=sa
jdbc.password=

# (3)
# Spring Batch schema initialize.
data-source.initialize.enabled=true
spring-batch.schema.script=classpath:org/springframework/batch/core/schema-h2.sql
terasoluna-batch.commit.script=classpath:org/terasoluna/batch/async/db/schema-
commit.sql
```

```

<!-- (3) -->
<jdbc:initialize-database data-source="adminDataSource"
                           enabled="${data-source.initialize.enabled:false}"
                           ignore-failures="ALL">
    <jdbc:script location="${spring-batch.schema.script}" />
    <jdbc:script location="${terasoluna-batch.commit.script}" />
</jdbc:initialize-database>

<!-- (4) -->
<bean id="adminDataSource" class="org.apache.commons.dbcp2.BasicDataSource"
      destroy-method="close"
      p:driverClassName="${admin.jdbc.driver}"
      p:url="${admin.jdbc.url}"
      p:username="${admin.jdbc.username}"
      p:password="${admin.jdbc.password}"
      p:maxTotal="10"
      p:minIdle="1"
      p:maxWaitMillis="5000"
      p:defaultAutoCommit="false"/>

<!-- (4) -->
<bean id="jobDataSource" class="org.apache.commons.dbcp2.BasicDataSource"
      destroy-method="close"
      p:driverClassName="${jdbc.driver}"
      p:url="${jdbc.url}"
      p:username="${jdbc.username}"
      p:password="${jdbc.password}"
      p:maxTotal="10"
      p:minIdle="1"
      p:maxWaitMillis="5000"
      p:defaultAutoCommit="false" />

<!-- (5) -->
<bean id="jobSqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean"
      p:dataSource-ref="jobDataSource" >
    <property name="configuration">
        <bean class="org.apache.ibatis.session.Configuration"
              p:localCacheScope="STATEMENT"
              p:lazyLoadingEnabled="true"
              p:aggressiveLazyLoading="false"
              p:defaultFetchSize="1000"
              p:defaultExecutorType="REUSE" />
    </property>
</bean>

```

```

<!-- (5) -->
<bean id="adminSqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean"
      p:dataSource-ref="adminDataSource" >
  <property name="configuration">
    <bean class="org.apache.ibatis.session.Configuration"
          p:localCacheScope="STATEMENT"
          p:lazyLoadingEnabled="true"
          p:aggressiveLazyLoading="false"
          p:defaultFetchSize="1000"
          p:defaultExecutorType="REUSE" />
  </property>
</bean>

```

#### データベース関連の設定における各要素の説明

項番	説明
(1)	pom.xmlでは利用するデータベースへの接続に使用するJDBCドライバの依存関係を定義する。 初期状態ではH2 Database(インメモリデータベース)とPostgreSQLが設定されているが、必要に応じて追加削除を行うこと。
(2)	JDBCドライバの接続設定をする。 - admin.jdbc.xxxはSpring BatchやTERASOLUNA Batch 5.xが利用する - jdbc.xxx～はジョブ個別が利用する
(3)	Spring BatchやTERASOLUNA Batch 5.xが利用するデータベースの初期化処理を実行するか否か、および、利用するスクリプトを定義する。 Spring Batchは <b>JobRepository</b> にアクセスするため、データベースが必須となる。 また、TERASOLUNA Batch 5.xは非同期実行( <b>DBポーリング</b> )にて ジョブ要求テーブルにアクセスするため、データベースが必須となる。 有効にするか否かは、以下を基準とするとよい。 - H2 Databaseを利用する場合は有効にする。無効にすると <b>JobRepository</b> やジョブ要求テーブルにアクセスできずエラーになる。 - H2 Databaseを利用しない場合は事故を予防するために無効にする。
(4)	データソースの設定をする。 必要に応じて接続数等をチューニングする。
(5)	MyBatisの挙動を設定する。 必要に応じてフェッチサイズ等をチューニングする。

#### 3.1.5. ジョブの作成

ジョブの作成方法は、以下を参照のこと。

- ・ チャンクモデルジョブの作成
- ・ タスクレットモデルジョブの作成

### 3.1.6. プロジェクトのビルドと実行

プロジェクトのビルドと実行について説明する。

#### 3.1.6.1. アプリケーションのビルド

プロジェクトのルートディレクトリに移動し、以下のコマンドを発行する。

ビルド(*Windows/Bash*)

```
$ mvn clean dependency:copy-dependencies -DoutputDirectory=lib package
```

これにより、以下が生成される。

- <ルートディレクトリ>/target/<archetypeId>-<version>.jar
  - 作成したバッチアプリケーションのJarが生成される
- <ルートディレクトリ>/lib/(依存Jarファイル)
  - 依存するJarファイル一式がコピーされる

試験環境や商用環境へ配備する際は、これらのJarファイルを任意のディレクトリにコピーすればよい。

#### 3.1.6.2. 環境に応じた設定ファイルの切替

プロジェクトのpom.xmlでは、初期値として以下のProfileを設定している。

## pom.xmlのProfiles設定

```
<profiles>
    <!-- Including application properties and log settings into package. (default) -->
    <profile>
        <id>IncludeSettings</id>
        <activation>
            <activeByDefault>true</activeByDefault>
        </activation>
        <properties>
            <exclude-property/>
            <exclude-log/>
        </properties>
    </profile>

    <!-- Excluding application properties and log settings into package. -->
    <profile>
        <id>ExcludeSettings</id>
        <activation>
            <activeByDefault>false</activeByDefault>
        </activation>
        <properties>
            <exclude-property>batch-application.properties</exclude-property>
            <exclude-log>logback.xml</exclude-log>
        </properties>
    </profile>
</profiles>
```

ここでは、環境依存となる設定ファイルを含めるかどうかを切替っている。この設定を活用して、環境配備の際に設定ファイルを別途配置することで環境差分を吸収することができる。また、これを応用して、試験環境と商用環境でJarに含める設定ファイルを変えることもできる。以下に、一例を示す。

## 環境ごとに設定ファイルを切替えるpom.xmlの記述例

```
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
    </resource>
    <resource>

<directory>${project.root.basedir}/${project.config.resource.directory.rdbms}</directory>
    </resource>
  </resources>
</build>

<profiles>
  <profile>
    <id>postgresql9-local</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <dependencies>
      <dependency>
        <groupId>org.postgresql</groupId>
        <artifactId>postgresql</artifactId>
        <scope>runtime</scope>
      </dependency>
    </dependencies>
    <properties>
      <project.config.resource.directory.rdbms>
config/rdbms/postgresql9/local</project.config.resource.directory.rdbms>
      </properties>
    </profile>
    <profile>
      <id>postgresql9-it</id>
      <dependencies>
        <dependency>
          <groupId>org.postgresql</groupId>
          <artifactId>postgresql</artifactId>
          <scope>runtime</scope>
        </dependency>
      </dependencies>
      <properties>
        <project.config.resource.directory.rdbms>
config/rdbms/postgresql9/it</project.config.resource.directory.rdbms>
        </properties>
      </profile>
    </profiles>
```

なお、MavenのProfileは以下の要領で、コマンド実行時に有効化することができる。

必要に応じて、複数Profileを有効化することもできる。必要に応じて、有効活用して欲しい。

#### MavenのProfileを有効化する例

```
$ mvn -P profile-1,profile-2
```

##### 3.1.6.2.1. アプリケーションの実行

前段でビルドした結果を元に、ジョブを実行する例を示す。

archetypeIdとversionはユーザの環境に応じて読み替えて欲しい。

コマンドプロンプト(*Windows*)

```
C:\xxxx> java -cp target\archetypeId-version.jar;lib\*^
org.springframework.batch.core.launch.support.CommandLineJobRunner^
META-INF/jobs/job01.xml job01
```

シェル(*Unix, Linux, ...*)

```
$ java -cp 'target/archetypeId-version.jar:lib/*' \
org.springframework.batch.core.launch.support.CommandLineJobRunner \
META-INF/jobs/job01.xml job01
```

*java*コマンドが返却する終了コードをハンドリングする必要性

実際のシステムでは、ジョブスケジューラからジョブを発行する際に*java*コマンドを直接発行するのではなく、*java*起動用のシェルスクリプトを挟んで起動することが一般的である。

これは*java*コマンド起動前の環境変数を設定するためや、*java*コマンドの終了コードをハンドリングするためである。この、*java*コマンドの終了コードをハンドリングは、以下を理由に常に行うことを推奨する。

- *java*コマンドの終了コードは正常:0、異常:1であるが、ジョブスケジューラはジョブの成功/失敗を終了コードの範囲で判断する。そのため、ジョブスケジューラの設定によっては、*java*コマンドは異常終了したのにもかかわらずジョブスケジューラは正常終了したと判断してしまう。
- OSやジョブスケジューラが扱うことができる終了コードは有限の範囲である。
  - OSやジョブスケジューラの仕様に応じて、ユーザにて使用する終了コードの範囲を定義することが重要である。
  - 一般的に、POSIX標準で策定されている0から255の間に収めることが多い。
    - TERASOLUNA Batch 5.xでは、正常:0、それ以外:255として終了コードを返却するよう設定している。



以下に、終了コードのハンドリング例を示す。

#### 終了コードのハンドリング例

```
#!/bin/bash

# ..omitted.

java -cp ...
RETURN_CODE=$?
if [ $RETURN_CODE = 1 ]; then
    return 255
else
    return $RETURN_CODE
fi
```

## 3.2. チャンクモデルジョブの作成

### 3.2.1. Overview

チャンクモデルジョブの作成方法について説明する。チャンクモデルのアーキテクチャについては、[Spring Batchのアーキテクチャ](#)を参照のこと。

ここでは、チャンクモデルジョブの構成要素について説明する。

#### 3.2.1.1. 構成要素

チャンクモデルジョブの構成要素を以下に示す。これらの構成要素をBean定義にて組み合わせることで1つのジョブを実現する。

#### チャンクモデルジョブの構成要素

項目番号	名称	役割	設定必須	実装必須
1	ItemReader	様々なリソースからデータを取得するためのインターフェース。 Spring Batchにより、扁平ファイルやデータベースを対象とした実装が提供されているため、ユーザにて作成する必要はない。	✓	-
2	ItemProcessor	入力から出力へデータを加工するためのインターフェース。 ユーザは必要に応じてこのインターフェースを <code>implements</code> し、ビジネスロジックを実装する。	-	-
3	ItemWriter	様々なリソースへデータを出力するためのインターフェース。 <code>ItemReader</code> と対になるインターフェースと考えてよい。 Spring Batchにより、扁平ファイルやデータベースのための実装が提供されているため、ユーザにて作成する必要はない。	✓	-

この表のポイントは以下である。

- 入力リソースから出力リソースへ単純にデータを移し替えるだけであれば、設定のみで実現できる。
- `ItemProcessor`は、必要が生じた際にのみ実装すればよい。

以降、これらの構成要素を用いたジョブの実装方法について説明する。

### 3.2.2. How to use

ここでは、実際にチャンクモデルジョブを実装する方法について、以下の順序で説明する。

- [ジョブの設定](#)
- [コンポーネントの実装](#)

### 3.2.2.1. ジョブの設定

Bean定義ファイルにて、チャンクモデルジョブを構成する要素の組み合わせ方を定義する。以下に例を示し、構成要素の繋がりを説明する。

Bean定義ファイルの例(チャンクモデル)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:batch="http://www.springframework.org/schema/batch"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:mybatis="http://mybatis.org/schema/mybatis-spring"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/batch
        http://www.springframework.org/schema/batch/spring-batch.xsd
        http://mybatis.org/schema/mybatis-spring
        http://mybatis.org/schema/mybatis-spring.xsd">

    <!-- (1) -->
    <import resource="classpath:META-INF/spring/job-base-context.xml"/>

    <!-- (2) -->
    <context:annotation-config/>

    <!-- (3) -->
    <context:component-scan
        base-package="org.terasoluna.batch.functionaltest.app.common" />

    <!-- (4) -->
    <mybatis:scan
        base-package="org.terasoluna.batch.functionaltest.app.repository.mst"
        factory-ref="jobSqlSessionFactory"/>

    <!-- (5) -->
    <bean id="reader"
        class="org.mybatis.spring.batch.MyBatisCursorItemReader" scope="step"
        p:queryId="org.terasoluna.batch.functionaltest.app.repository.mst.CustomerRepository.findAll"
        p:sqlSessionFactory-ref="jobSqlSessionFactory"/>

    <!-- (6) -->
    <!-- Item Processor -->
    <!-- Item Processor in order that based on the Bean defined by the annotations,
        not defined here -->
```

```

<!-- (7) -->
<bean id="writer"
      class="org.springframework.batch.item.file.FlatFileItemWriter"
      scope="step"
      p:resource="file:#{jobParameters[outputFile]}">
    <property name="lineAggregator">
      <bean
        class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
        <property name="fieldExtractor">
          <bean
            class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"
            p:names="customerId,customerName,customerAddress,customerTel,chargeBranchId"/>
        </property>
      </bean>
    </property>
  </bean>

<!-- (8) -->
<batch:job id="jobCustomerList01" job-repository="jobRepository" > <!-- (9) -->
  <batch:step id="jobCustomerList01.step01" > <!-- (10) -->
    <batch:tasklet transaction-manager="jobTransactionManager" > <!-- (11) -->
      <batch:chunk reader="reader"
                    processor="processor"
                    writer="writer"
                    commit-interval="10" /> <!-- (12) -->
    </batch:tasklet>
  </batch:step>
</batch:job>
</beans>

```

#### ItemProcessor実装クラスの設定

```

@Component("processor") // (6)
public class CustomerProcessor implement ItemProcessor<Customer, Customer> {
  // omitted
}

```

項番	説明
(1)	TERASOLUNA Batch 5.xを利用する際に、常に必要なBean定義を読み込む設定をインポートする。
(2)	アノテーションによるBean定義の有効化を行う。ItemProcessorやListenerなどを実装する際に、(3)と合わせて利用する。
(3)	コンポーネントスキャン対象のベースパッケージを設定する。アノテーションによるBean定義を行う場合は、(2)と合わせて利用する。
(5)	ItemReaderの設定。 ItemReaderの詳細は、 <a href="#">データベースアクセス</a> 、 <a href="#">ファイルアクセス</a> を参照のこと。

項目番	説明
(6)	ItemProcessorは、(2),(3)によりアノテーションにて定義することができ、Bean定義ファイルで定義する必要がない。
(7)	ItemWriterの設定。 ItemWriterの詳細は、 <a href="#">データベースアクセス</a> 、 <a href="#">ファイルアクセス</a> を参照のこと。
(8)	ジョブの設定。 id属性に設定する値は、1つのバッチアプリケーションに含まれる全ジョブの範囲において、一意とすること。
(9)	JobRepositoryの設定。 <code>job-repository</code> 属性に設定する値は、特別な理由がない限り <code>jobRepository</code> 固定とすること。 これにより、すべてのジョブが1つのJobRepositoryで管理できる。 <code>jobRepository</code> のBean定義は、(1)により解決する。
(10)	ステップの設定。 id属性に設定する値は、1つのジョブ内で一意とすること。 接頭辞として(8)で設定したid属性を付加した形式にし、 ステップもジョブと同様にバッチアプリケーションに含まれる全ジョブの範囲において一意とすると、ログ出力や異常発生時の特定をはじめ、様々な場面で有効活用できる。 よって、特別な理由がない限り <code>&lt;ジョブのid&gt;.&lt;step名&gt;</code> とすること。
(11)	タスクレットの設定。 <code>transaction-manager</code> 属性に設定する値は、特別な理由がない限り <code>jobTransactionManager</code> 固定とすること。 これにより、(12)の <code>commit-interval</code> ごとにトランザクションが管理される。 詳細については、 <a href="#">トランザクション制御</a> を参照のこと。 <code>jobTransactionManager</code> のBean定義は、(1)により解決する。
(12)	チャンクモデルジョブの設定。 <code>reader</code> 、 <code>processor</code> 、 <code>writer</code> の各属性に対し、前段までで定義した <code>ItemReader</code> 、 <code>ItemProcessor</code> 、 <code>ItemWriter</code> のBeanIDを指定する。 <code>commit-interval</code> 属性に1チャンクあたりの入力データ件数を設定する。

#### *commit-interval*のチューニング

`commit-interval`はチャンクモデルジョブにおける、性能上のチューニングポイントである。



前述の例では10件としているが、利用できるマシンリソースやジョブの特性によって適切な件数は異なる。複数のリソースにアクセスしてデータを加工するジョブであれば10件から100件程度で処理スループットが頭打ちになることもある。一方、入出力リソースが1:1対応しておりデータを移し替える程度のジョブであれば5000件や10000件でも処理スループットが伸びることがある。

ジョブ実装時の`commit-interval`は100件程度で仮置きしておき、その後に実施した性能測定の結果に応じてジョブごとにチューニングするとよい。

### 3.2.2.2. コンポーネントの実装

ここでは主に、ItemProcessorを実装する方法について説明する。

他のコンポーネントについては、以下を参照のこと。

- ItemReader、ItemWriter
  - データベースアクセス、ファイルアクセス
- Listener
  - リスナー

### 3.2.2.2.1. ItemProcessorの実装

ItemProcessorの実装方法を説明する。

ItemProcessorは、以下のインターフェースが示すとおり、入力リソースから取得したデータ 1件を元に、出力リソースに向けたデータ 1件を作成する役目を担う。つまり、ItemProcessorはデータ 1件に対するビジネスロジックを実装する箇所、と言える。

*ItemProcessor*インターフェース

```
public interface ItemProcessor<I, O> {
    O process(I item) throws Exception;
}
```

なお、インターフェースが示す **I** と **O** は以下のとおり同じ型でも異なる型でもよい。同じ型であれば入力データを一部修正することを意味し、異なる型であれば入力データを元に出力データを生成することを意味する。

*ItemProcessor*実装例(入出力が同じ型)

```
@Component
public class AmountUpdateItemProcessor implements
    ItemProcessor<SalesPlanDetail, SalesPlanDetail> {

    @Override
    public SalesPlanDetail process(SalesPlanDetail item) throws Exception {
        item.setAmount(new BigDecimal("1000"));
        return item;
    }
}
```

## ItemProcessor実装例(入出力が異なる型)

```
@Component
public class UpdateItemFromDBProcessor implements
    ItemProcessor<SalesPerformanceDetail, SalesPlanDetail> {

    @Inject
    CustomerRepository customerRepository;

    @Override
    public SalesPlanDetail process(SalesPerformanceDetail readItem) throws Exception {
        Customer customer = customerRepository.findOne(readItem.getCustomerId());

        SalesPlanDetail writeItem = new SalesPlanDetail();
        writeItem.setBranchId(customer.getChargeBranchId());
        writeItem.setYear(readItem.getYear());
        writeItem.setMonth(readItem.getMonth());
        writeItem.setCustomerId(readItem.getCustomerId());
        writeItem.setAmount(readItem.getAmount());
        return writeItem;
    }
}
```

### ItemProcessorからnullを返却することの意味



ItemProcessorからnullを返却することは、当該データを後続処理(Writer)に渡さないことを意味し、言い換えるとデータをフィルタすることになる。これは、入力データの妥当性検証を実施する上で有効活用できる。詳細については、[入力チェック](#)を参照のこと。

### ItemProcessorの処理スループットをあげるには

前述した実装例のように、ItemProcessorの実装クラスではDBやファイルを始めとしたリソースにアクセスしなければならないことがある。ItemProcessorは入力データ1件ごとに実行されるため、I/Oが少しでも発生するとジョブ全体では大量のI/Oが発生することになる。そのため、極力I/Oを抑えることが処理スループットをあげる上で重要なとなる。



1つの方法として、後述のListenerを活用することで事前に必要なデータをメモリ上に確保しておき、ItemProcessorにおける処理の大半を、CPU/メモリ間で完結するように実装する手段がある。ただし、1ジョブあたりのメモリを大量に消費することにも繋がるので、何でもメモリ上に確保すればよいわけではない。I/O回数やデータサイズを元に、メモリに格納するデータを検討すること。

この点については、[データの入出力](#)でも紹介する。

複数のItemProcessorを同時に利用する

汎用的なItemProcessorを用意し、個々のジョブに適用したい場合は、Spring Batchが提供するCompositeItemProcessorを利用し連結することで実現できる。

CompositeItemProcessorによる複数ItemProcessorの連結



```
<bean id="processor"
      class="org.springframework.batch.item.support.CompositeItemProcessor">
    <property name="delegates">
      <list>
        <ref bean="commonItemProcessor"/>
        <ref bean="businessLogicItemProcessor"/>
      </list>
    </property>
</bean>
```

delegates属性に指定した順番に処理されることに留意すること。

## 3.3. タスクレットモデルジョブの作成

### 3.3.1. Overview

タスクレットモデルジョブの作成方法について説明する。タスクレットモデルのアーキテクチャについては、[Spring Batchのアーキテクチャ](#)を参照のこと。

#### 3.3.1.1. 構成要素

タスクレットモデルジョブでは、複数の構成要素は登場しない。

`org.springframework.batch.core.step.tasklet.Tasklet`を実装し、Bean定義で設定するのみである。また、発展的な実装手段としてチャンクモデルの構成要素である`ItemReader`や`ItemWriter`をコンポーネントとして使うことも可能である。

### 3.3.2. How to use

ここでは、実際にタスクレットモデルジョブを実装する方法について、以下の順序で説明する。

- [ジョブの設定](#)
- [Taskletの実装](#)

#### 3.3.2.1. ジョブの設定

Bean定義ファイルにて、タスクレットモデルジョブを定義する。以下に例を示す。

## Bean定義ファイルの例(タスクレットモデル)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:batch="http://www.springframework.org/schema/batch"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/batch
http://www.springframework.org/schema/batch/spring-batch.xsd">

    <!-- (1) -->
    <import resource="classpath:META-INF/spring/job-base-context.xml"/>

    <!-- (2) -->
    <context:annotation-config/>

    <!-- (3) -->
    <context:component-scan
        base-package="org.terasoluna.batch.functionaltest.app.common"/>

    <!-- (4) -->
    <batch:job id="jobSimpleJob" job-repository="jobRepository"> <!-- (5) -->
        <batch:step id="simpleJob.step01"> <!-- (6) -->
            <batch:tasklet transaction-manager="jobTransactionManager"
                           ref="simpleJobTasklet"/> <!-- (7) -->
        </batch:step>
    </batch:job>

</beans>
```

## Tasklet実装クラスの例

```
package org.terasoluna.batch.functionaltest.app.common;

@Component // (3)
public class SimpleJobTasklet implements Tasklet {
    // omitted
}
```

項番	説明
(1)	TERASOLUNA Batch 5.xを利用する際に、常に必要なBean定義を読み込む設定をインポートする。
(2)	アノテーションによるBean定義の有効化を行う。(3)と合わせて利用する。

項目番	説明
(3)	コンポーネントスキャン対象のベースパッケージを設定する。(2)と合わせて利用する。タスクレットモデルはアノテーションによるBean定義を基本とし、Tasklet実装クラスのBean定義はXML上では不要とする。
(4)	ジョブの設定。 id属性に設定する値は、1つのバッチアプリケーションに含まれる全ジョブの範囲において、一意とすること。
(5)	<b>JobRepository</b> の設定。 <b>job-repository</b> 属性に設定する値は、特別な理由がない限り <b>jobRepository</b> 固定とすること。これにより、すべてのジョブが1つの <b>JobRepository</b> で管理できる。 <b>jobRepository</b> のBean定義は、(1)により解決する。
(6)	ステップの設定。 id属性に設定する値は、1ジョブ内で一意とすること。 接頭辞として(8)で設定したid属性を付加した形式にする。 こうすることでステップもジョブと同様にバッチアプリケーションに含まれる全ジョブの範囲において一意となり、ログ出力や異常発生時の特定をはじめ、様々な場面で有効活用できる。 よって、特別な理由がない限り<ジョブのid>.<step名>とすること。
(7)	タスクレットの設定。 <b>transaction-manager</b> 属性に設定する値は、特別な理由がない限り <b>jobTransactionManager</b> 固定とすること。 これにより、タスクレット全体の処理が1つのトランザクションで管理される。 詳細については、 <a href="#">トランザクション制御</a> を参照のこと。 <b>jobTransactionManager</b> のBean定義は、(1)により解決する。 また、 <b>ref</b> 属性に設定する値は、(3)により解決するBean名となる。 ここでは、Tasklet実装クラス名 <b>SimpleJobTasklet</b> の先頭を小文字にした <b>simpleJobTasklet</b> となる。

アノテーション利用時のBean名



@Componentアノテーション利用時のBean名は、デフォルトでは  
`org.springframework.context.annotation.AnnotationBeanNameGenerator`を通じて生成される。命名ルールを確認したいときは、本クラスのJavadocを参照するとよい。

### 3.3.2.2. Taskletの実装

まずはシンプルな実装で概要を理解し、次にチャンクモデルのコンポーネントを利用する実装へと進む。

以下の順序で説明する。

- シンプルなTaskletの実装
- チャンクモデルのコンポーネントを利用するTasklet実装

### 3.3.2.3. シンプルなTaskletの実装

ログを出力するのみのTasklet実装を通じ、最低限のポイントを説明する。

## シンプルなTasklet実装クラスの例

```
package org.terasoluna.batch.functionaltest.app.common;

// omitted

@Component
public class SimpleJobTasklet implements Tasklet { // (1)

    private static final Logger logger =
        LoggerFactory.getLogger(SimpleJobTasklet.class);

    @Override
    public RepeatStatus execute(StepContribution contribution,
        ChunkContext chunkContext) throws Exception { // (2)
        logger.info("called tasklet."); // (3)
        return RepeatStatus.FINISHED; // (4)
    }
}
```

項番	説明
(1)	org.springframework.batch.core.step.tasklet.Taskletインターフェースをimplementsして実装する。
(2)	Taskletインターフェースが定義するexecuteメソッドを実装する。引数のStepContribution, ChunkContextは必要に応じて利用するが、ここでは説明を割愛する。
(3)	任意の処理を実装する。ここではINFOログを出力している。
(4)	Taskletの処理が完了したかどうかを返却する。 常にreturn RepeatStatus.FINISHED;と明示する。

### 3.3.2.4. チャンクモデルのコンポーネントを利用するTasklet実装

Spring Batch では、Tasklet実装の中でチャンクモデルの各種コンポーネントを利用するに言及していない。TERASOLUNA Batch 5.xでは、以下のような状況に応じてこれを選択可能してよい。

- 複数のリソースを組み合わせながら処理するため、チャンクモデルの形式に沿いにくい
- チャンクモデルでは処理が複数箇所に実装することになるため、タスクレットモデルの方が全体像を把握しやすい
- リカバリをシンプルにするため、チャンクモデルの中間コミットではなく、タスクレットモデルの一括コミットを使いたい

以下に、チャンクモデルのコンポーネントであるItemReaderやItemWriterを利用するTasklet実装について説明する。

## チャンクモデルのコンポーネントを利用するTasklet実装例1

```
@Component()
@Scope("step") // (1)
public class SalesPlanChunkTranTask implements Tasklet {

    @Inject
    @Named("detailCSVReader") // (2)
    ItemStreamReader<SalesPlanDetail> itemReader; // (3)

    @Inject
    SalesPlanDetailRepository repository; // (4)

    @Override
    public RepeatStatus execute(StepContribution contribution,
        ChunkContext chunkContext) throws Exception {

        SalesPlanDetail item;

        try {
            itemReader.open(chunkContext.getStepContext().getStepExecution()
                .getExecutionContext()); // (5)

            while ((item = itemReader.read()) != null) { // (6)

                // do some processes.

                repository.create(item); // (7)
            }
        } finally {
            itemReader.close(); // (8)
        }
        return RepeatStatus.FINISHED;
    }
}
```

## Bean定義例1

```
<!-- omitted -->
<import resource="classpath: META-INF/spring/job-base-context.xml"/>

<context:annotation-config/>

<context:component-scan
    base-package="org.terasoluna.batch.functionaltest.app.plan" />
<context:component-scan
    base-package="org.terasoluna.batch.functionaltest.ch05.transaction.component" />

<!-- (9) -->
<mybatis:scan
    base-package="org.terasoluna.batch.functionaltest.app.repository.plan"
    factory-ref="jobSqlSessionFactory"/>

<!-- (10) -->
<bean id="detailCSVReader"
    class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
    p:resource="#{jobParameters[inputFile]}">
    <property name="lineMapper">
        <bean class="org.springframework.batch.item.mapping.DefaultLineMapper">
            <property name="lineTokenizer">
                <bean
                    class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
                    p:names="branchId,year,month,customerId,amount"/>
            </property>
            <property name="fieldSetMapper">
                <bean
                    class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"
                    p:targetType="org.terasoluna.batch.functionaltest.app.model.plan.SalesPlanDetail"/>
                </property>
            </bean>
        </property>
    </bean>
</bean>

<!-- (11) -->
<batch:job id="createSalesPlanChunkTranTask" job-repository="jobRepository">
    <batch:step id="createSalesPlanChunkTranTask.step01">
        <batch:tasklet transaction-manager="jobTransactionManager"
            ref="salesPlanChunkTranTask"/>
    </batch:step>
</batch:job>
```

項番	説明
(1)	本クラス内で利用するItemReaderのBeanスコープに合わせ、stepスコープとする。

項目番	説明
(2)	入力リソース(この例ではフラットファイル)へのアクセスはItemReaderを通じて行う。ここでは、 <code>detailCSVReader</code> というBean名を指定するが、わかりやすさのためなので任意とする。
(3)	ItemReaderのサブインターフェースである、ItemStreamReaderとして型を定義する。これは、(5), (8)のリソースオープン/クローズを実装する必要があるためである。後ほど補足する。
(4)	出力リソース(この例ではデータベース)へのアクセスはMyBatisのMapperを通じて行う。ここでは、簡単のためMapperを直接利用している。常にItemWriterを用いる必要はない。もちろん、MyBatisBatchItemWriterを用いてもよい。
(5)	入力リソースをオープンする。
(6)	入力リソース全件を逐次ループ処理する。 ItemReader#readは、入力データがすべて読み取り末端に到達した場合、nullを返却する。
(7)	データベースへ出力する。
(8)	リソースは必ずクローズすること。 なお、例外処理は必要に応じて実装すること。 ここで例外が発生した場合、タスクレット全体のトランザクションがロールバックされ、例外のスタックトレースを出力し、ジョブが異常終了する。
(9)	データベースへ出力するため、 <code>mybatis:scan</code> の設定を追加する。詳細はここでは割愛する。
(10)	ファイルから入力するため、FlatFileItemReaderのBean定義を追加する。詳細はここでは割愛する。
(11)	各種コンポーネントはアノテーションによって解決するため、 <a href="#">シンプルなTaskletの実装</a> の場合と同様となる。

### スコープの統一について

Tasklet実装クラスと、InjectするBeanのスコープは、同じスコープに統一すること。

たとえば、FlatFileItemReaderが引数から入力ファイルパスを受け取る場合にはBeanスコープをstepにする必要がある。この時、Tasklet実装クラスのスコープもstepにする必要がある。



仮にTasklet実装クラスのスコープをsingletonとしたケースを説明する。この時、アプリケーション起動時のApplicationContext生成時にTasklet実装クラスをインスタンス化した後、FlatFileItemReaderのインスタンスを解決してInjectしようとする。しかし、FlatFileItemReaderはstepスコープでありstep実行時に生成するためまだ存在しない。結果、Tasklet実装クラスをインスタンス化できないと判断しApplicationContext生成に失敗してしまう。

`@Inject`を付与するフィールドの型について  
利用する実装クラスに応じて、以下のいずれかとする。

- ItemReader/ItemWriter
  - 対象となるリソースへのオープン・クローズを実施する必要がない場合に利用する。
- ItemSteamReader/ItemStreaWriter
  - 対象となるリソースへのオープン・クローズを実施する必要がある場合に利用する。



必ずjavadocを確認してどちらを利用するか判断すること。以下に代表例を示す。

*FlatFileItemReader/Writer*の場合

ItemSteamReader/ItemStreaWriterにて扱う

*MyBatisCursorItemReader*の場合

ItemStreamReaderにて扱う

*MyBatisBatchItemWriter*の場合

ItemWriterにて扱う

もう1つの例として、`ItemReader`と`ItemWriter`を同時に用いた場合を示す。

チャンクモデルのコンポーネントを利用するTasklet実装例2

```
@Component
@Scope("step")
public class SalesPerformanceTasklet implements Tasklet {

    @Inject
    ItemStreamReader<SalesPerformanceDetail> reader;

    @Inject
    ItemWriter<SalesPerformanceDetail> writer; // (1)

    int chunkSize = 10; // (2)

    @Override
    public RepeatStatus execute(StepContribution contribution,
                               ChunkContext chunkContext) throws Exception {

        try {
            reader.open(chunkContext.getStepContext().getStepExecution()
                       .getExecutionContext());

            List<SalesPerformanceDetail> items = new ArrayList<>(chunkSize); // (2)
            SalesPerformanceDetail item = null;
```

```
do {
    // Pseudo operation of ItemReader
    for (int i = 0; i < chunkSize; i++) { // (3)
        item = reader.read();
        if (item == null) {
            break;
        }
        // Pseudo operation of ItemProcessor
        // do some processes.

        items.add(item);
    }

    // Pseudo operation of ItemWriter
    if (!items.isEmpty()) {
        writer.write(items); // (4)
        items.clear();
    }
} while (item != null);
} finally {
    try {
        reader.close();
    } catch (Exception e) {
        // do nothing.
    }
}

return RepeatStatus.FINISHED;
}
}
```

## Bean定義例2

```
<!-- omitted -->
<import resource="classpath: META-INF/spring/job-base-context.xml"/>

<context:annotation-config/>
<context:component-scan
    base-package="org.terasoluna.batch.functionaltest.app.common,
        org.terasoluna.batch.functionaltest.app.performance,
        org.terasoluna.batch.functionaltest.ch06.exceptionhandling"/>
<mymbatis:scan
    base-package="org.terasoluna.batch.functionaltest.app.repository.performance"
    factory-ref="jobSqlSessionFactory"/>

<bean id="detailCSVReader"
    class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
    p:resource="file:#{jobParameters[inputFile]}">
    <property name="lineMapper">
        <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <property name="lineTokenizer">
                <bean
                    class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
                    p:names="branchId,year,month,customerId,amount"/>
            </property>
            <property name="fieldSetMapper">
                <bean
                    class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"
                    p:targetType="org.terasoluna.batch.functionaltest.app.model.performance.SalesPerformanceDetail"/>
            </property>
        </bean>
    </property>
</bean>

<!-- (1) -->
<bean id="detailWriter"
    class="org.mybatis.spring.batch.MyBatisBatchItemWriter"
    p:statementId="org.terasoluna.batch.functionaltest.app.repository.performance.SalesPerformanceDetailRepository.create"
    p:sqlSessionTemplate-ref="batchModeSqlSessionTemplate"/>

<batch:job id="jobSalesPerfTasklet" job-repository="jobRepository">
    <batch:step id="jobSalesPerfTasklet.step01">
        <batch:tasklet ref="salesPerformanceTasklet"
            transaction-manager="jobTransactionManager"/>
    </batch:step>
</batch:job>
```

項目番	説明
(1)	<code>ItemWriter</code> の実装として <code>MyBatisBatchItemWriter</code> を利用する。
(2)	<code>ItemWriter</code> は一定件数をまとめて出力する。 ここでは10件ごとに処理し出力する。
(3)	チャンクモデルの動作にそって、 <code>read</code> → <code>process</code> → <code>read</code> → <code>process</code> →...→ <code>write</code> となるようにする。
(4)	<code>ItemWriter</code> を通じてまとめて出力する。

`ItemReader`や`ItemWriter`の実装クラスを利用するかどうかは都度判断してほしいが、 ファイルアクセスは`ItemReader`や`ItemWriter`の実装クラスを利用するとよいだろう。 それ以外のデータベースアクセス等は無理に使う必要はない。 性能向上のために使えばよい。

## 3.4. チャンクモデルとタスクレットモデルの使い分け

ここでは、チャンクモデルとタスクレットモデルの使い分けについて、それぞれの特徴を整理することで説明する。なお、説明において、以降の章で詳細な説明をする事項もあるため、適宜対応する章を参照して欲しい。

また、以降の内容は考え方の一例として捉えて欲しい。制約や推奨事項ではない。ユーザやシステムの特性に応じてジョブを作成する際の参考にしてほしい。

以下に、チャンクモデルとタスクレットモデルの主要な違いについて列挙する。

### チャンクモデルとタスクレットモデルの比較

項目	チャンク	タスクレット
構成要素	<code>ItemReader</code> , <code>ItemProcessor</code> , <code>ItemWriter</code> の3つに分割する。	<code>Tasklet</code> の1つに集約する。
トランザクション	一定件数で中間コミットを発行しながら処理することが基本となる。一括コミットはできない。 処理対象データ件数に依らず一定のマシンリソースで処理できる。 処理途中でエラーが発生すると未処理データと処理済データが混在する。	全体で一括コミットにて処理することが基本となる。中間コミットはユーザにて実装する必要がある。 処理対象データが大量になると、マシンリソースが枯渇する恐れがある。 処理途中でエラーが発生すると未処理データのみにロールバックされる。
リスター ト	件数ベースのリストアートができる。	件数ベースのリストアートはできない。

これを踏まえて、以下にそれぞれを使い分ける例をいくつか紹介する。

#### リカバリを限りなくシンプルにしたい

エラーとなったジョブは対象のジョブをリランするのみで復旧したい場合など、リカバリをシンプルにしたい時はタスクレットモデルを選択するとよい。

チャンクモデルでは処理済データをジョブ実行前の状態に戻したり、未処理データのみ処理するようジョブを予め作りこんでおいたり、といった対処が必要となる。

#### 処理の内容をまとめたい

1ジョブ1クラスなど、ジョブの見通しを優先したい場合はタスクレットを選択するとよい。

#### 大量のデータを安定して処理したい

1000万件など、一括処理するとリソースに影響する件数を対象とする際はチャンクモデルを活用するか検討するとよい。これは中間コミットによって安定させることを意味する。タスクレットモデルでも中間コミットを打つことが可能だが、チャンクモデルの方がシンプルな実装になる可能性がある。

#### エラー後の復旧は件数ベースリストアートとしたい

バッチウィンドウがシビアであり、エラーとなったデータ以降から再開したい場合に、Spring Batchが提供する件数ベースリストアートを活用するときは、チャンクモデルを選択する必要がある。これにより、個々のジョブでその仕組を作りこむ必要がなくなる。

チャンクモデルとタスクレットモデルは、併用することが基本である。  
バッチシステム内のジョブすべてをどちらかのモデルでのみ実装する必要はない。  
システム全体のジョブがもつ特性を踏まえて、一方のモデルを基本とし、状況に応じ  
てもう一方のモデルを使うことは自然である。



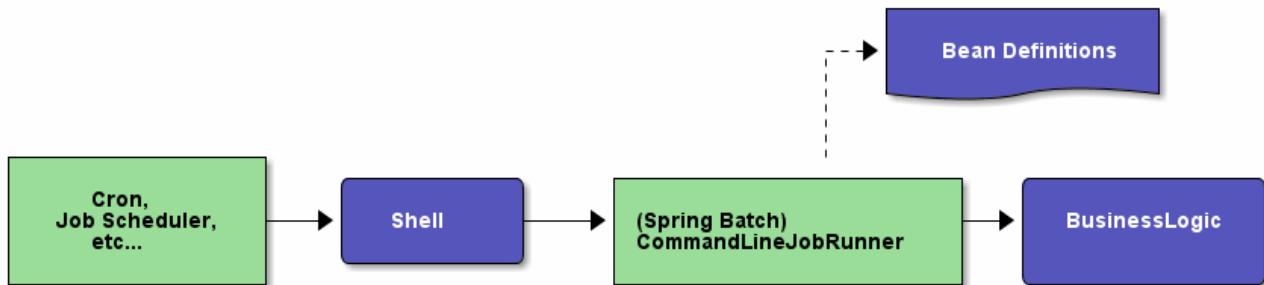
たとえば、大部分は処理件数や処理時間に余裕があるならばタスクレットモデルを基  
本とし、極少数の大量件数を処理するジョブはチャンクモデルを選択する、といった  
ことは自然といえる。

# Chapter 4. ジョブの起動

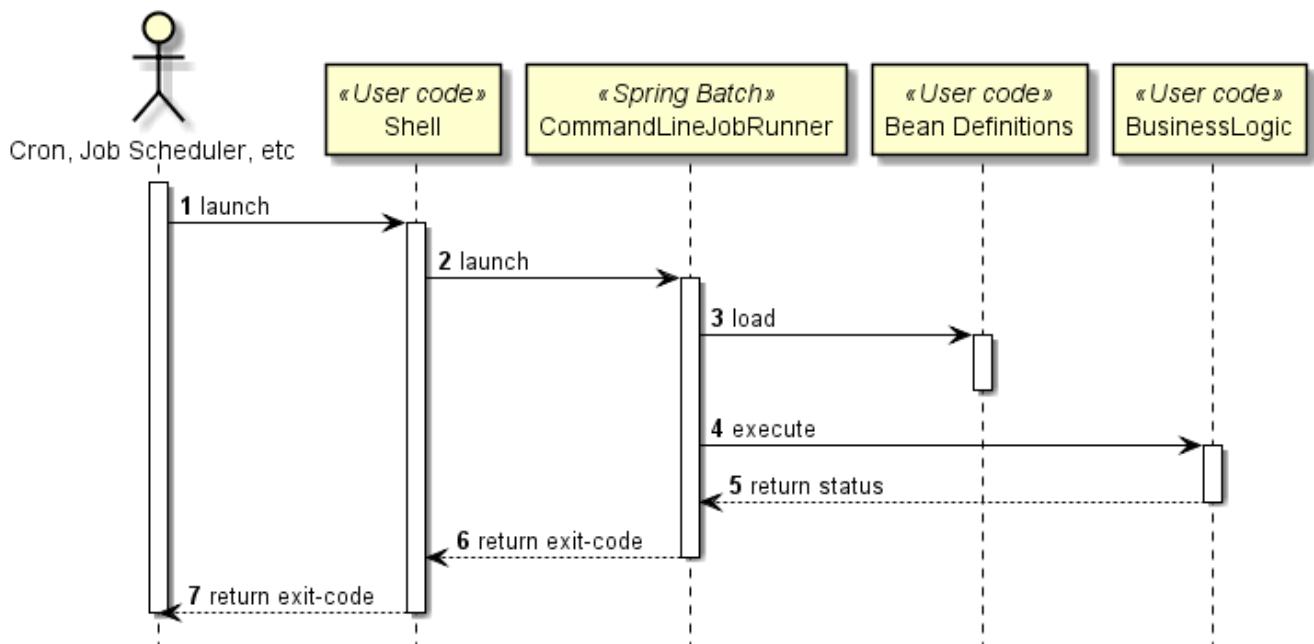
## 4.1. 同期実行

### 4.1.1. Overview

同期実行について説明する。同期実行とは、ジョブスケジューラなどによりシェルを介して新規プロセスとして起動し、ジョブの実行結果を呼び出し元に返却する実行方法である。



同期実行の概要



同期実行の流れ

本機能は、チャンクモデルとタスクレットモデルとで同じ使い方になる。

### 4.1.2. How to use

[CommandLineJobRunner](#)によってジョブを起動する方法を説明する。

なお、アプリケーションのビルドや実行については、[プロジェクトの作成](#)を参照のこと。また、起動パラメータの指定方法や活用方法については、[ジョブの起動パラメータ](#)を参照のこと。これらと本節の説明は一部重複するが、同期実行の要素に注目して説明する。

#### 4.1.2.1. 実行方法

TERASOLUNA Batch 5.xにおいて、同期実行は Spring Batch が提供する **CommandLineJobRunner** によって実現する。**CommandLineJobRunner** は、以下の要領にて java コマンドを発行することで起動する。

*CommandLineJobRunner* の構文

```
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner <jobPath>
<options> <jobIdentifier> <jobParameters>
```

引数にて指定する項目

指定する項目	説明	必須
jobPath	起動するジョブの設定を記述した Bean 定義ファイルのパス。 classpath からの相対パスにて指定する。	✓
options	起動する際の各種オプション(停止、リスタートなど)を指定する。	
jobIdentifier	ジョブの識別子として、Bean 定義上のジョブ名、もしくはジョブを実行後のジョブ実行 ID を指定する。通常はジョブ名を指定する。ジョブ実行 ID は停止やリスタートの際にのみ指定する。	✓
jobParameters	ジョブの引数を指定する。指定は <b>key=value</b> 形式となる。	

以下に、必須項目のみを指定した場合の実行例を示す。

*CommandLineJobRunner* の実行例1

```
$ java -cp 'target/archetypeId-version.jar:lib/*' \ # (1)
org.springframework.batch.core.launch.support.CommandLineJobRunner \ # (2)
META-INF/jobs/job01.xml job01 # (3)
```

Bean 定義の設定(抜粋)

```
<batch:job id="job01" job-repository="jobRepository"> <!-- (3) -->
  <batch:step id="job01.step01">
    <batch:tasklet transaction-manager="jobTransactionManager">
      <batch:chunk reader="employeeReader"
                    processor="employeeProcessor"
                    writer="employeeWriter" commit-interval="10" />
    </batch:tasklet>
  </batch:step>
</batch:job>
```

設定内容の項目一覧

項番	説明
(1)	java コマンドを実行する際に、バッチアプリケーションの jar と、依存する jar を <b>classpath</b> に指定する。ここではコマンド引数で指定しているが、環境変数等を用いてもよい。

項目番	説明
(2)	起動するクラスに、 <code>CommandLineJobRunner</code> をFQCNで指定する。
(3)	<code>CommandLineJobRunner</code> に沿って、起動引数を渡す。ここでは、 <code>jobPath</code> と <code>jobIdentifier</code> としてジョブ名の2つを指定している。

次に、任意項目として起動パラメータを指定した場合の実行例を示す。

#### `CommandLineJobRunner`の実行例2

```
$ java -cp 'target/archetypeId-version.jar:lib/*' \
    org.springframework.batch.core.launch.support.CommandLineJobRunner \
    META-INF/jobs/setupJob.xml setupJob target=server1 outputFile=/tmp/result.csv #
(1)
```

#### 設定内容の項目一覧

項目番	説明
(1)	ジョブの起動パラメータとして、 <code>target=server1</code> と <code>outputFile=/tmp/result.csv</code> を指定している。

#### 4.1.2.2. 任意オプション

`CommandLineJobRunner`の構文で示した任意のオプションについて補足する。

`CommandLineJobRunner`では以下の4つの起動オプションが使用できる。ここでは個々の説明は他に委ねることとし、概要のみ説明する。

`-restart`

失敗したジョブを再実行する。詳細は、[処理の再実行](#)を参照のこと。

`-stop`

実行中のジョブを停止する。詳細は、[ジョブの管理](#)を参照のこと。

`-abandon`

停止されたジョブを放棄する。放棄されたジョブは再実行不可となる。TERASOLUNA Batch 5.xでは、このオプションを活用するシーンがないため、説明を割愛する。

`-next`

過去に一度実行完了したジョブを再度実行する。ただし、TERASOLUNA Batch 5.xでは、このオプションを利用しない。

なぜなら、TERASOLUNA Batch 5.xでは、Spring Batchのデフォルトである「同じパラメータで起動したジョブは同一ジョブとして認識され、同一ジョブは1度しか実行できない」という制約を回避しているためである。

詳細は[パラメータ変換クラスについて](#)にて説明する。

また、本オプションを利用するには、`JobParametersIncrementer`というインターフェースの実装クラスが必要だが、TERASOLUNA Batch 5.xでは設定を行っていない。

そのため、本オプションを指定して起動すると、必要なBean定義が存在しないためエラーとなる。

## 4.2. ジョブの起動パラメータ

### 4.2.1. Overview

本節では、ジョブの起動パラメータ(以降、パラメータ)の利用方法について説明する。

本機能は、チャンクモデルとタスクレットモデルとで同じ使い方になる。

パラメータは、以下のような実行環境や実行タイミングに応じてジョブの動作を柔軟に切替える際に使用する。

- 処理対象のファイルパス
- システムの運用日時

パラメータを与える方法は、以下のとおりである。

1. コマンドライン引数から与える
2. ファイルから標準入力ヘリダイレクトする

指定したパラメータは、Bean定義やSpring管理下のJavaで参照できる。

### 4.2.2. How to use

#### 4.2.2.1. パラメータ変換クラスについて

Spring Batchでは、受け取ったパラメータを以下の流れで処理する。

1. `JobParametersConverter`の実装クラスが`JobParameters`に変換する。
2. Bean定義やSpring管理下のJavaにて`JobParameters`からパラメータを参照する。

パラメータ変換クラスの実装クラスについて

前述した`JobParametersConverter`の実装クラスは複数提供されている。以下にそれぞれの特徴を示す。

- `DefaultJobParametersConverter`
  - パラメータのデータ型を指定することができる(String、Long、Date、Doubleの4種類)。
- `JsrJobParametersConverter`
  - パラメータのデータ型を指定することができない(Stringのみ)。
  - パラメータにジョブ実行を識別するID(RUN\_ID)を`jsr_batch_run_id`という名称で自動的に付与する。
    - RUN\_IDは、ジョブが実行される都度増分する。増分は、データベースのSEQUENCE(名称は`JOB_SEQ`となる)を利用するため、重複することがない。
    - Spring Batchでは、同じパラメータで起動したジョブは同一ジョブとして認識され、同一ジョブは1度しか実行できない、という仕様がある。これに対し、`jsr_batch_run_id`という名称のパラメータを一意な値で付加することにより、別のジョブと認識する仕組みとなっている。詳細は、[Spring Batchのアーキテクチャ](#)を参照すること。

Spring BatchではBean定義で使用するJobParametersConverterの実装クラスを指定しない場合、DefaultJobParametersConverterが使用される。

しかし、TERASOLUNA Batch 5.xでは以下の理由によりDefaultJobParametersConverterは採用しない。

- 1つのジョブを同じパラメータによって、異なるタイミングで起動することは一般的である。
- 起動時刻のタイムスタンプなどを指定し、異なるジョブとして管理することも可能だが、それだけのためにジョブパラメータを指定するのは煩雑である。
- DefaultJobParametersConverterはパラメータに対しデータ型を指定することができるが、型変換に失敗した場合のハンドリングが煩雑になる。

TERASOLUNA Batch 5.xでは、JsrJobParametersConverterを利用することで、ユーザが意識することなく自動的にRUN\_IDを付与している。この仕組みにより、ユーザから見ると同一ジョブをSpring Batchとしては異なるジョブとして扱っている。

パラメータ変換クラスの設定について

TERASOLUNA Batch 5.xでは、予めlaunch-context.xmlにてJsrJobParametersConverterを使用するよう設定している。

そのためTERASOLUNA Batch 5.xを推奨設定で使用する場合はJobParametersConverterの設定を行う必要はない。

META-INF|spring|launch-context.xml

```
<bean id="jobParametersConverter"
      class="org.springframework.batch.core.jsr.JsrJobParametersConverter"
      c:dataSource-ref="adminDataSource" />

<bean id="jobOperator"
      class="org.springframework.batch.core.launch.support.SimpleJobOperator"
      p:jobRepository-ref="jobRepository"
      p:jobRegistry-ref="jobRegistry"
      p:jobExplorer-ref="jobExplorer"
      p:jobParametersConverter-ref="jobParametersConverter"
      p:jobLauncher-ref="jobLauncher" />
```

以降はJsrJobParametersConverterを利用する前提で説明する。

#### 4.2.2.2. コマンドライン引数から与える

まず、もっとも基本的な、コマンドライン引数から与える方法について説明する。

パラメータの付与

コマンドライン引数としてCommandLineJobRunnerの第3引数以降に<パラメータ名>=<値>形式で列挙する。

パラメータの個数や長さは、Spring BatchやTERASOLUNA Batch 5.xにおいては制限がない。しかし、OSにはコマンド引数の長さに制限がある。

そのため、あまりに大量の引数が必要な場合は、[ファイルから標準入力ヘリダイレクトする](#)や[パラメータとプロパティの併用](#)などの方法を活用すること。

## コマンドライン引数としてパラメータを設定する例

```
# Execute job
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
  JobDefined.xml JOBID param1=abc outputFileName=/tmp/result.csv
```

### パラメータの参照

以下のように、Bean定義またはJavaで参照することができる。

- Bean定義で参照する
  - `#{jobParameters[xxx]}`で参照可能
- Javaで参照する
  - `@Value("#{jobParameters[xxx]}")`で参照可能

*JobParameters*を参照するBeanのスコープはStepスコープでなければならない

*JobParameters*を参照する際は、参照するBeanのスコープをStepスコープとする必要がある。これは、*JobParameters*を参照する際に、Spring Batchの**late binding**という仕組みを使用しているためである。

**late binding**とはその名のとおり、遅延して値を設定することである。Spring Frameworkの*ApplicationContext*は、デフォルトでは各種Beanのプロパティを解決してから*ApplicationContext*のインスタンスを生成する。Spring Batchでは*ApplicationContext*のインスタンスを生成する時にはプロパティを解決せず、各種Beanが必要になった際にプロパティを解決する機能をもつ。これが遅延という言葉が意味することである。この機能により、Spring Batch自体の実行に必要な*ApplicationContext*を生成し実行した後に、パラメータに応じて各種Beanの振る舞いを切替えることが可能となる。

なお、StepスコープはSpring Batch独自のスコープであり、Stepの実行ごとに新たなインスタンスが生成される。また、**late binding**による値の解決は、Bean定義においてSpEL式を用いることで可能となる。

Stepスコープの指定では@*StepScope*アノテーションは使用できない

Spring Batchでは、Stepスコープを指定するアノテーションとして@*StepScope*が提供されているが、これはJavaConfigにおいてのみ使用できるアノテーションである。

そのため、TERASOLUNA Batch 5.xにおけるStepスコープの指定は以下のいずれかの方法で行う。

1. Bean定義では、Beanに`scope="step"`を付与する。
2. Javaでは、クラスに`@Scope("step")`を付与する。

## コマンドライン引数で与えたパラメータをBean定義で参照する例

```
<!-- (1) -->
<bean id="reader"
    class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
    p:resource="file:#{jobParameters[inputFile]}> <!-- (2) -->
<property name="lineMapper">
    <!-- omitted settings -->
</property>
</bean>
```

### 設定内容の項目一覧

項目番	説明
(1)	beanタグにscope属性としてスコープを指定する。
(2)	参照するパラメータを指定する。

## コマンドライン引数で与えたパラメータをJavaで参照する例

```
@Component
@Scope("step") // (1)
public class ParamRefInJavaTasklet implements Tasklet {

    /**
     * Holds a String type value
     */
    @Value("#{jobParameters[str]}") // (2)
    private String str;

    // omitted execute()
}
```

### 設定内容の項目一覧

項目番	説明
(1)	クラスに@Scopeアノテーションを付与してスコープを指定する。
(2)	@Valueアノテーションを使用して参照するパラメータを指定する。

### 4.2.2.3. ファイルから標準入力ヘリダイレクトする

ファイルから標準入力ヘリダイレクトする方法について説明する。

#### パラメータを定義するファイルの作成

パラメータは下記のようにファイルに定義する。

## params.txt

```
param1=abc  
outputFile=/tmp/result.csv
```

パラメータを定義したファイルを標準入力ヘリダイレクトする

コマンドライン引数としてパラメータを定義したファイルをリダイレクトする。

### 実行方法

```
# Execute job  
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \  
  JobDefined.xml JOBID < params.txt
```

パラメータの参照

パラメータの参照方法は[コマンドライン引数から与える](#)方法と同様である。

#### 4.2.2.4. パラメータのデフォルト値を設定する

パラメータを任意とした場合、以下の形式でデフォルト値を設定することができる。

- `#{jobParameters[パラメータ名] ?: デフォルト値}`

ただし、パラメータを使用して値を設定している項目であるということは、デフォルト値もパラメータと同様に環境や実行タイミングによって異なる可能性がある。

まずは、デフォルト値をソースコード上にハードコードをする方法を説明する。しかし、後述の[パラメータとプロパティの併用](#)を活用する方が適切なケースが多いため、合わせて参照すること。

デフォルト値を設定したパラメータの参照

該当するパラメータが設定されなかった場合にデフォルト値に設定した値が参照される。

コマンドライン引数で与えたパラメータをBean定義で参照する例

```
<!-- (1) -->  
<bean id="reader"  
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"  
      p:resource="file:#{jobParameters[inputFile] ?: /input/sample.csv}"> <!-- (2)  
-->  
  <property name="lineMapper">  
    // omitted settings  
  </property>  
</bean>
```

### 設定内容の項目一覧

項目番	説明
(1)	beanタグにscope属性としてスコープを指定する。

項目番	説明
(2)	参照するパラメータを指定する。 デフォルト値に <code>/output/result.csv</code> を設定している。

コマンドライン引数で与えたパラメータをJavaで参照する例

```
@Component
@Scope("step") // (1)
public class ParamRefInJavaTasklet implements Tasklet {

    /**
     * Holds a String type value
     */
    @Value("#{jobParameters[str] ?: xyz}") // (2)
    private String str;

    // omitted execute()
}
```

#### 設定内容の項目一覧

項目番	説明
(1)	クラスに <code>@Scope</code> アノテーションを付与してスコープを指定する。
(2)	<code>@Value</code> アノテーションを使用して参照するパラメータを指定する。 デフォルト値に <code>xyz</code> を設定している。

#### 4.2.2.5. パラメータの妥当性検証

オペレーションミスや意図しない挙動を防ぐために、ジョブの起動時にパラメータの妥当性検証が必要となる場合もある。

パラメータの妥当性検証はSpring Batchが提供する`JobParametersValidator`を活用することで実現可能である。

パラメータはItemReader/ItemProcessor/ItemWriterといった様々な場所で参照するため、ジョブの起動直後に妥当性検証が行われる。

パラメータの妥当性を検証する方法は2つあり、検証の複雑度によって異なる。

- 簡易な妥当性検証

- 適用例
  - 必須パラメータが設定されていることの検証
  - 意図しないパラメータが設定されていないことの検証
- 使用するバリデータ
  - Spring Batchが提供している`DefaultJobParametersValidator`

- 複雑な妥当性検証

- 適用例
  - 数値の範囲検証やパラメータ間の相関チェックなどの複雑な検証
  - Spring Batchが提供しているDefaultJobParametersValidatorにて実現不可能な検証
- 使用するバリデータ
  - JobParametersValidatorを自作で実装したクラス

簡易な妥当性検証および複雑な妥当性検証の妥当性を検証する方法についてそれぞれ説明する。

#### 4.2.2.5.1. 簡易な妥当性検証

Spring BatchはJobParametersValidatorのデフォルト実装として、DefaultJobParametersValidatorを提供している。

このバリデータでは設定により以下を検証することができる。

- 必須パラメータが設定されていること
- 必須または任意パラメータ以外のパラメータが指定されていないこと

以下に定義例を示す。

DefaultJobParametersValidatorを使用する妥当性検証の定義

```
<!-- (1) -->
<bean id="jobParametersValidator"
      class="org.springframework.batch.core.job.DefaultJobParametersValidator">
  <property name="requiredKeys"> <!-- (2) -->
    <list>
      <value>jsr_batch_run_id</value> <!-- (3) -->
      <value>inputFileName</value>
      <value>outputFileName</value>
    </list>
  </property>
  <property name="optionalKeys"> <!-- (4) -->
    <list>
      <value>param1</value>
      <value>param2</value>
    </list>
  </property>
</bean>

<batch:job id="jobUseDefaultJobParametersValidator" job-repository="jobRepository">
  <batch:step id="jobUseDefaultJobParametersValidator.step01">
    <batch:tasklet ref="sampleTasklet" transaction-manager="jobTransactionManager"/>
  </batch:step>
  <batch:validator ref="jobParametersValidator"/> <!-- (5) -->
</batch:job>
```

設定内容の項目一覧

項目番	説明
(1)	<code>DefaultJobParametersValidator</code> のBeanを定義する。
(2)	必須パラメータはプロパティ <code>requiredKeys</code> に設定する。 listタグを使用して必須パラメータのパラメータ名を複数指定できる。
(3)	必須パラメータに <code>jsr_batch_run_id</code> を設定する。 TERASOLUNA Batch 5.xでは、 <code>DefaultJobParametersValidator</code> を使用する場合はこの設定が必須である。 必須となる理由は後述する。
(4)	任意パラメータはプロパティ <code>optionalKeys</code> に設定する。 listタグを使用して任意パラメータのパラメータ名を複数指定できる。
(5)	jobタグ内にvalidatorタグを使用してジョブにバリデータを適用する。

TERASOLUNA Batch 5.xでは省略できない必須パラメータ

TERASOLUNA Batch 5.xではパラメータ変換に`JsrJobParametersConverter`を採用しているため、以下のパラメータが常に設定される。

- `jsr_batch_run_id`

そのため、`requiredKeys`には、`jsr_batch_run_id`を必ず含めること。  
詳細な説明は、[パラメータ変換クラスについて](#)を参照すること。

パラメータの定義例

```
!<bean id="jobParametersValidator"
  class="org.springframework.batch.core.job.DefaultJobParametersValidator">
  <property name="requiredKeys">
    <list>
      <value>jsr_batch_run_id</value> <!-- mandatory -->
      <value>inputFileName</value>
      <value>outputFileName</value>
    </list>
  </property>
  <property name="optionalKeys">
    <list>
      <value>param1</value>
      <value>param2</value>
    </list>
  </property>
</bean>
```

`DefaultJobParametersValidator`を使用した場合のOKケースとNGケース

`DefaultJobParametersValidator`にて検証可能な条件の理解を深めるため、検証結果がOKとなる場合とNGとなる場合の例を示す。

## DefaultJobParametersValidator定義例

```
<bean id="jobParametersValidator"
  class="org.springframework.batch.core.job.DefaultJobParametersValidator"
  p:requiredKeys="outputFileName"
  p:optionalKeys="param1"/>
```

### NGケース1

```
# Execute job
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
  JobDefined.xml JOBID param1=aaa
```

必須パラメータ**outputFile**が設定されていないためNGとなる。

### NGケース2

```
# Execute job
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
  JobDefined.xml JOBID outputFileName=/tmp/result.csv param2=aaa
```

必須パラメータ、任意パラメータのどちらにも指定されていないパラメータ**param2**が設定されたためNGとなる。

### OKケース1

```
# Execute job
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
  JobDefined.xml JOBID param1=aaa outputFileName=/tmp/result.csv
```

必須および任意として指定されたパラメータが設定されているためOKとなる。

### OKケース2

```
# Execute job
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
  JobDefined.xml JOBID fileoutputfilename=/tmp/result.csv
```

必須パラメータが設定されているためOKとなる、任意パラメータは設定されてもよい。

#### 4.2.2.5.2. 複雑な妥当性検証

**JobParametersValidator**インターフェースの実装を自作することで、要件に応じたパラメータの検証を実現することができる。

**JobParametersValidator**クラスは以下の要領で実装する。

- **JobParametersValidator**クラスを実装し、validateメソッドをオーバーライドする

- validateメソッドは以下の要領で実装する
  - JobParametersから各パラメータを取得し検証する
    - 検証の結果がOKである場合には、何もする必要はない
    - 検証の結果がNGである場合には、JobParametersInvalidExceptionをスローする

JobParametersValidatorクラスの実装例を示す。ここでは、strで指定された文字列の長さが、numで指定された数値以下であることを検証している。

*JobParametersValidator*インターフェースの実装例

```
public class ComplexJobParametersValidator implements JobParametersValidator { // (1)
    @Override
    public void validate(JobParameters parameters) throws
        JobParametersInvalidException {
        Map<String, JobParameter> params = parameters.getParameters(); // (2)

        String str = params.get("str").getValue().toString(); // (3)
        int num = Integer.parseInt(params.get("num").getValue().toString()); // (4)

        if(str.length() > num){
            throw new JobParametersInvalidException(
                "The str must be less than or equal to num. [str:"
                + str + "][num:" + num + "]"); // (5)
        }
    }
}
```

設定内容の項目一覧

項目番	説明
(1)	JobParametersValidatorクラスを実装しvalidateメソッドをオーバーライドする。
(2)	パラメータはJobParameters型で引数として受ける。 parameters.getParameters()とすることで、Map形式で取得することでパラメータの参照が容易になる。
(3)	keyを指定してパラメータを取得する。
(4)	パラメータをint型へ変換する。String型以外を扱う場合は適宜変換を行うこと。
(5)	パラメータstrの文字列長がパラメータnumの値を超えている場合に妥当性検証結果NGとしている。

## ジョブの定義例

```
<batch:job id="jobUseComplexJobParametersValidator" job-repository="jobRepository">
    <batch:step id="jobUseComplexJobParametersValidator.step01">
        <batch:tasklet ref="sampleTasklet" transaction-manager=
"jobTransactionManager"/>
    </batch:step>
    <batch:validator> <!-- (1) -->
        <bean
class="org.terasoluna.batch.functionaltest.ch04.jobparameter.ComplexJobParametersValidator"/>
    </batch:validator>
</batch:job>
```

### 設定内容の項目一覧

項目番	説明
(1)	jobタグ内にvalidatorタグを使用してジョブにバリデータを適用する。

非同期起動時におけるパラメータの妥当性検証について

非同期起動方式(DBポーリングやWebコンテナ)でも、同様にジョブ起動時に検証することは可能だが、以下のようなタイミングでジョブを起動する前に検証することが望ましい。

- DBポーリング
  - ジョブ要求テーブルへのINSERT前
- Webコンテナ
  - Controller呼び出し時(@Validatedを付与する)



非同期起動の場合、結果は別途確認する必要が生じるため、パラメータ設定ミスのような場合は早期にエラーを応答し、ジョブの要求をリジェクトすることが望ましい。

また、この時の妥当性検証において、`JobParametersValidator`を使う必要はない。ジョブ要求テーブルへINSERTする機能や、Webコンテナ上のControllerは多くの場合Spring Batchに依存していないはずであり、`JobParametersValidator`を使用するためだけにSpring Batchに依存することは避けた方がよい。

### 4.2.3. How to extends

#### 4.2.3.1. パラメータとプロパティの併用

Spring BatchのベースであるSpring Frameworkには、プロパティ管理の機能が備わっており、環境変数やプロパティファイルに設定した値を扱うことができる。詳細は、TERASOLUNA Server 5.x 開発ガイドラインの[プロパティ管理](#)を参照すること。

プロパティとパラメータを組み合わせることで、大部分のジョブに共通的な設定をプロパティファイルに行なったうえで、一部をパラメータで上書きするといったことが可能になる。

パラメータとプロパティが解決されるタイミングについて

前述のとおり、パラメータとプロパティは、機能を提供するコンポーネントが異なる。

Spring Batchはパラメータ管理の機能をもち、Spring Frameworkはプロパティ管理の機能をもつ。

この差は記述方法の差に現れている。

- Spring Batchがもつ機能の場合
  - `#{{jobParamaters[xxx]}}`
- Spring Frameworkがもつ機能の場合
  - `@Value("${xxx}")`



また、それぞれの値が解決されるタイミングが異なる。

- Spring Batchがもつ機能の場合
  - ApplicationContextを生成後、ジョブを実行するタイミングで設定される。
- Spring Frameworkがもつ機能の場合
  - ApplicationContextの生成時に設定される。

よって、Spring Batchによるパラメータの値が優先される結果になる。

この点を念頭におくと、組み合わせる際に応用が効くため両者を区別して扱うこと。

以降、プロパティとパラメータを組み合わせて設定する方法について説明する。

環境変数による設定に加えて、コマンドライン引数で追加設定する場合

環境変数による設定に加えて、コマンドライン引数を使用してパラメータを設定する方法を説明する。

Bean定義においても同様に参照可能である。

環境変数に加えてコマンドライン引数でパラメータを設定する例

```
# Set environment variables
$ export env1=aaa
$ export env2=bbb

# Execute job
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
  JobDefined.xml JOBID param3=ccc outputFile=/tmp/result.csv
```

## Javaにおいて環境変数とパラメータを参照する例

```
@Value("${env1}") // (1)
private String param1;

@Value("${env2}") // (1)
private String param2;

private String param3;

@Value("#{jobParameters[param3]}") // (2)
public void setParam3(String param3) {
    this.param3 = param3;
}
```

### 設定内容の項目一覧

項目番	説明
(1)	@Valueアノテーションを使用して参照する環境変数を指定する。 参照する際の形式は\${環境変数名}である。
(2)	@Valueアノテーションを使用して参照するパラメータを指定する。 参照する際の形式は#{jobParameters[パラメータ名]}である。

### 環境変数をデフォルトとする場合の例

```
# Set environment variables
$ export env1=aaa

# Execute job
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
  JobDefined.xml JOBID param1=bbb outputFile=/tmp/result.csv
```

## Javaにおいて環境変数をデフォルト値としてパラメータを参照する例

```
@Value("#{jobParameters[param1] ?: '${env1}'}") // (1)
public void setParam1(String param1) {
    this.param1 = param1;
}
```

### 設定内容の項目一覧

項目番	説明
(1)	環境変数をデフォルト値として@Valueアノテーションを使用して参照するパラメータを指定する。 パラメータが設定されなかった場合、環境変数の値が設定される。

## 誤ったデフォルト値の設定方法

以下の要領で定義した場合、コマンドライン引数からparam1を設定しない場合に、env1の値が設定されてほしいにも関わらず、param1にnullが設定されてしまうため注意すること。

### 誤ったデフォルト値の設定方法例



```
@Value("${env1}")
private String param1;

@Value("#{jobParameters[param1]}")
public void setParam1(String param1) {
    this.param1 = param1;
}
```

## 4.3. 非同期実行(DBポーリング)

### 4.3.1. Overview

DBポーリングによるジョブ起動について説明をする。

本機能は、チャンクモデルとタスクレットモデルとで同じ使い方になる。

#### 4.3.1.1. DBポーリングによるジョブの非同期実行とは

非同期実行させたいジョブを登録する専用のテーブル(以降、ジョブ要求テーブル)を一定周期で監視し、登録された情報を元にジョブを非同期実行することをいう。

TERASOLUNA Batch 5.xでは、テーブルを監視しジョブを起動するモジュールを非同期バッチデーモンという名称で定義する。非同期バッチデーモンは1つのJavaプロセスとして稼働し、1ジョブごとにプロセス内のスレッドを割り当てて実行する。

##### 4.3.1.1.1. TERASOLUNA Batch 5.xが提供する機能

TERASOLUNA Batch 5.xは、以下の機能を非同期実行(DBポーリング)として提供する。

#### 非同期実行(DBポーリング)の機能一覧

機能	説明
非同期バッチデーモン機能	ジョブ要求テーブルポーリング機能を常駐実行させる機能
ジョブ要求テーブルポーリング機能	ジョブ要求テーブルに登録された情報にもとづいてジョブを非同期実行する機能。 ジョブ要求テーブルのテーブル定義も合わせて提供する。

#### 利用前提

ジョブ要求テーブルでは、ジョブ要求のみを管理する。要求されたジョブの実行状況および結果は、[JobRepository](#)に委ねる。これら2つを通じてジョブのステータスを管理することを前提としている。

また、[JobRepository](#)にインメモリデータベースを使用すると、非同期バッチデーモン停止後に[JobRepository](#)がクリアされ、ジョブの実行状況および結果を参照できない。そのため、[JobRepository](#)には永続性が担保されているデータベースを使用することを前提とする。

#### インメモリデータベースの使用

[JobRepository](#)を参照せずにジョブ実行結果の成否を得る手段がある場合、インメモリデータベースで運用するケースも考えられる。

インメモリデータベースで長期連続運用をする場合、メモリリソースを大量消費してジョブ実行に悪影響を及ぼす可能性がある。

つまり、インメモリデータベースは、長期連続運用するには向かず、定期的に再起動する運用が望ましい。

それでも長期連続運用で利用したい場合は、定期的に[JobRepository](#)からデータを削除するなどのメンテナンス作業が必須である。

再起動する場合は、初期化を有効にしておけば再起動時に再作成されるため、メンテナンスは不要である。初期化については、[データベース関連の設定](#)参照。



#### 4.3.1.1.2. 活用シーン

非同期実行(DBポーリング)を活用するシーンを以下にいくつか示す。

##### 活用シーン一覧

活用シーン	説明
ディレード処理	オンライン処理と連携して、即時に完了する必要がなく、かつ、時間がかかる処理をジョブとして切り出したい場合。
処理時間が短いジョブの連続実行	1ジョブあたり数秒～数十秒の処理を連続実行する場合。非同期実行(DBポーリング)を活用することで、1ジョブごとにJavaプロセスの起動・終了によるリソースの圧迫を回避できる。また、起動・終了処理を割愛することに繋がるためジョブの実行時間を短縮することが可能となる。
大量にあるジョブの集約	処理時間が短いジョブの連続実行と同様である。

##### 非同期実行(Webコンテナ)と使い分けるポイント

非同期実行(Webコンテナ)と使い分けるポイントを以下に示す。



- ・バッチ処理にWebAPサーバを導入することにハードルがある
- ・可用性を担保する際に、データベースのみを考慮すればよい
  - ・その代わり、データベースにアクセスが集中するため、非同期実行(Webコンテナ)ほどスケールしない可能性がある

##### Spring Batch Integerationを採用しない理由

Spring Batch Integerationを利用して同様の機能を実現することは可能である。



しかし、Spring Batch Integerationを使用すると非同期実行以外の要素も含めた技術要素の理解・取得が必要となる。それにより、本機能の理解/活用/カスタマイズが難しくなるのを避けるため、Spring Batch Integerationの適用は見送っている。

##### 非同期実行(DBポーリング)での注意点

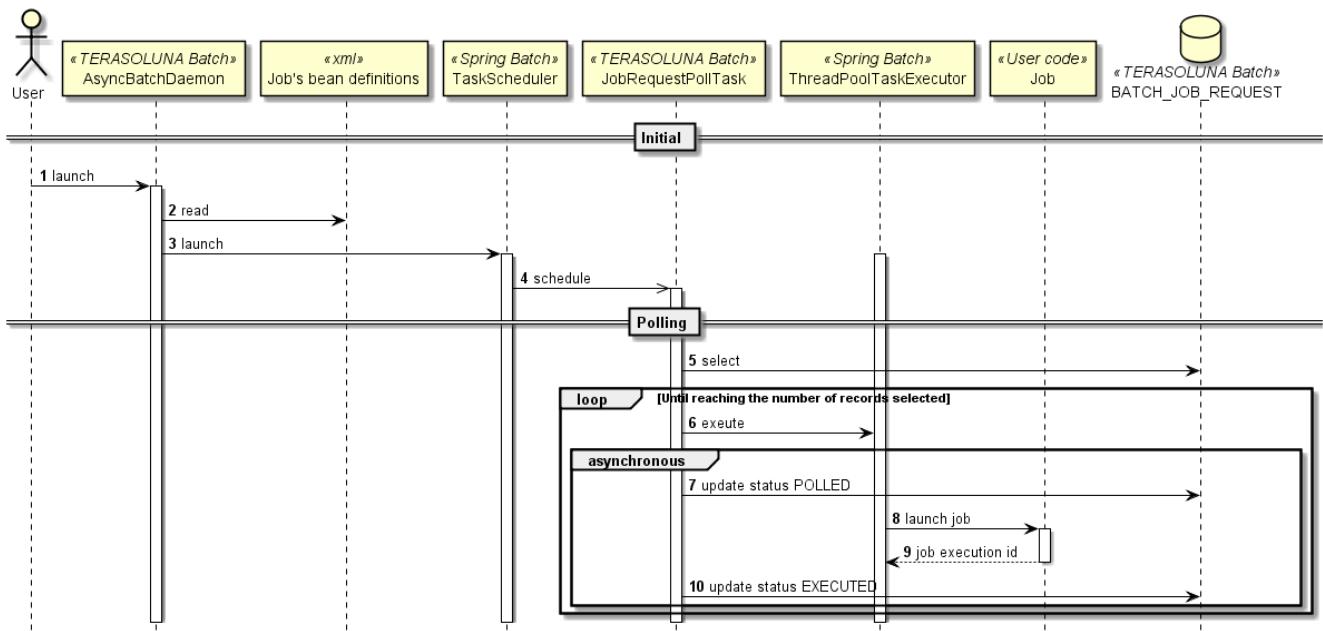


1ジョブあたり数秒にも満たない超ショートバッチを大量に実行する場合、JobRepositoryも含めてデータベースへのアクセスが都度発生する。この点に起因する性能劣化もあり得るため、超ショートバッチの大量処理は、非同期実行(DBポーリング)には向いていない。本機能を利用する際はこの点を踏まえ、目標性能を満たせるか十分に検証をすること。

## 4.3.2. Architecture

### 4.3.2.1. DBポーリングの処理シーケンス

DBポーリングの処理シーケンスについて説明する。



DBポーリングの処理シーケンス図

1. **AsyncBatchDeamon**をshなどから起動する。
2. **AsyncBatchDeamon**は、起動時にジョブを定義したBean定義ファイルをすべて読み込む。
3. **AsyncBatchDeamon**は、一定間隔でポーリングするために**TaskScheduler**を起動する。
  - **TaskScheduler**は、一定間隔で特定の処理を起動する。
4. **TaskScheduler**は、**JobRequestPollTask**(ジョブ要求テーブルをポーリングする処理)を起動する。
5. **JobRequestPollTask**は、ジョブ要求テーブルからポーリングステータスが未実行(INIT)のレコードを取得する。
  - 一定件数をまとめて取得する。デフォルトは3件。
  - 対象のレコードが存在しない場合は、一定間隔を空けて再度ポーリングを行う。デフォルトは5秒間隔。
6. **JobRequestPollTask**は、レコードの情報にもとづいて、ジョブをスレッドに割り当てて実行する。
7. **JobRequestPollTask**は、ジョブ要求テーブルのポーリングステータスをポーリング済み(POLLED)へ更新する。
  - ジョブの同時実行数に達している場合は、取得したレコードから起動できないレコードを破棄し、次回ポーリング処理時にレコードを再取得する。
8. スレッドに割り当てられたジョブは、**JobOperator**によりジョブを開始する。
9. 実行したジョブのジョブ実行ID(Job execution id)を取得する。
10. **JobRequestPollTask**は、ジョブ実行時に取得したジョブ実行IDにもとづいて、ジョブ要求テーブルのポーリングステータスをジョブ実行済み(EXECUTED)に更新する。

## 処理シーケンスの補足

Spring Batchのリファレンスでは、[JobLauncher](#)に[AsyncTaskExecutor](#)を設定することで非同期実行が実現できることを示している。しかし、この方法を採用すると[AsyncTaskExecutor](#)がジョブ実行が出来ない状態を検知できない。これは、ジョブに割り当てられるスレッドがない時などに発生し、その結果以下の事象に繋がる可能性がある。



- ・ジョブが実行できないにも関わらず、ジョブの起動をしようとして続け不要な処理をしてしまう
- ・スレッドが空いたタイミングによっては、ポーリングした順番にジョブが起動せず、ジョブ要求テーブル上ランダムに起動するよう見えてしまう

この事象を回避するため前述の処理シーケンスとなっている。

### 4.3.2.2. ポーリングするテーブルについて

非同期実行(DBポーリング)でポーリングを行うテーブルについて説明する。

以下データベースオブジェクトを必要とする。

- ・ジョブ要求テーブル(必須)
- ・ジョブシーケンス(データベース製品によっては必須)
  - ・データベースがカラムの自動採番に対応していない場合に必要となる。

#### 4.3.2.2.1. ジョブ要求テーブルの構造

以下に、TERASOLUNA Batch 5.xが対応しているデータベース製品のうち、PostgreSQLの場合を示す。他のデータベースについては、TERASOLUNA Batch 5.xのjarに同梱されているDDLを参照してほしい。

*batch\_job\_request (PostgreSQLの場合)*

カラム名	データ型	制約	説明
job_seq_id	bigserial (別途シーケンスを定義する場合は、bigintとする)	NOT NULL PRIMARY KEY	ポーリング時に実行するジョブの順序を決める番号。 データベースの自動採番機能を利用。
job_name	varchar(100)	NOT NULL	実行するジョブ名。 ジョブ実行時の必須パラメータ。
job_parameter	varchar(2000)	-	実行するジョブに渡すパラメータ。 单一パラメータの書式は同期実行と同じだが、複数パラメータを指定する場合は、同期型実行の空白区切りとは異なり、各パラメータをカンマ区切り(下記参照)にする必要がある。 {パラメータ名}={パラメータ値},{パラメータ名}={パラメータ値}...

カラム名	データ型	制約	説明
job_execution_id	bigint	-	ジョブ実行時に払い出されるID。 このIDをキーにしてJobRepositoryを参照する。
polling_status	varchar(10)	NOT NULL	ポーリング処理状況。 INIT : 未実行 POLLLED: ポーリング済み EXECUTED : ジョブ実行済み
create_date	TIMESTAMP	NOT NULL	ジョブ要求のレコードを登録した日時。
update_date	TIMESTAMP	-	ジョブ要求のレコードを更新した日時。

DDLは以下のとおり。

```
CREATE TABLE IF NOT EXISTS batch_job_request (
    job_seq_id bigserial PRIMARY KEY,
    job_name varchar(100) NOT NULL,
    job_parameter varchar(200),
    job_execution_id bigint,
    polling_status varchar(10) NOT NULL,
    create_date timestamp NOT NULL,
    update_date timestamp
);
```

#### 4.3.2.2.2. ジョブ要求シーケンスの構造

データベースがカラムの自動採番に対応していない場合は、シーケンスによる採番が必要になる。

以下に、TERASOLUNA Batch 5.xが対応しているデータベース製品のうち、PostgreSQLの場合を示す。他のデータベースについては、TERASOLUNA Batch 5.xのjarに同梱されているDDLを参照してほしい。

DDLは以下のとおり。

```
CREATE SEQUENCE batch_job_request_seq MAXVALUE 9223372036854775807 NO CYCLE;
```



PostgreSQLはカラムの自動採番に対応しているため、TERASOLUNA Batch 5.xのjarに同梱されているDDLにジョブ要求シーケンスは定義されていない。シーケンスの最大値を変更したい場合などに、`job_seq_id`のデータ型を`bigserial`から`bigint`に変更した上で、ジョブ要求シーケンスを定義すると良い。

#### 4.3.2.2.3. ポーリングステータス(polling\_status)の遷移パターン

ポーリングステータスの遷移パターンを下表に示す。

#### ポーリングステータスの遷移パターン一覧

遷移元	遷移先	説明
INIT	INIT	同時実行数に達して、ジョブの実行を拒否された場合はステータスの変更はない。 次回ポーリング時にポーリング対象のレコードとなる。
INIT	POLLED	ジョブの起動に成功した時に遷移する。 ジョブを実行している時のステータス。
POLLED	EXECUTED	ジョブの実行が終了した時に遷移する。

#### 4.3.2.3. ジョブの起動について

ジョブの起動方法について説明をする。

TERASOLUNA Batch 5.xのジョブ要求テーブルポーリング機能内部では、Spring Batchから提供されているJobOperatorのstartメソッドでジョブを起動する。

TERASOLUNA Batch 5.xでは、非同期実行(DBポーリング)で起動したジョブのリストアートは、コマンドラインからの実行をガイドしている。そのため、JobOperatorにはstart以外にもrestartなどの起動メソッドがあるが、satrtメソッド以外は使用していない。

startメソッドの引数

jobName

ジョブ要求テーブルのjob\_nameに登録した値を設定する。

jobParametrers

ジョブ要求テーブルのjob\_parametersに登録した値を設定する。

#### 4.3.2.4. DBポーリング処理で異常が発生した場合について

DBポーリング処理で異常が発生した場合について説明する。

##### 4.3.2.4.1. データベース接続障害

障害が発生した時点で行われていた処理別に振る舞いを説明する。

ジョブ要求テーブルからのレコード取得時

- JobRequestPollTaskはエラーとなるが、次回のポーリングにてJobRequestPollTaskが再実行される。

ポーリングステータスをINITからPOLLEDに変更する間

- JobOperatorによるジョブ実行前にJobRequestPollTaskはエラー終了する。ポーリングステータスは、INITのままになる。
- 接続障害回復後に行われるポーリング処理では、ジョブ要求テーブルに変更がないため実行対象となり、次回ポーリング時にジョブが実行される。

ポーリングステータスをPOLLEDからEXECUTEDに変更する間

- JobRequestPollTaskは、ジョブ実行IDをジョブ要求テーブルに更新できずにエラー終了する。ポーリングステータスは、POLLEDのままになる。

- 接続障害回復後に行われるポーリング処理の対象外となり、障害時のジョブは実行されない。
- ジョブ要求テーブルからジョブ実行IDを知ることができないため、ジョブの最終状態をログやJobRepositoryから判断し、必要に応じてジョブの再実行など回復処理を行う。

**JobRequestPollTask**で例外が発生しても、即座に自動復旧しようとはしない。以下に理由を示す。



- JobRequestPollTask**は、一定間隔で起動するため、これに委ねることで(即座ではないが)自動復旧できる。
- 障害発生時に即座にリトライしても回復できるケースは稀であり、かえってリトライにより負荷を発生してしまう可能性がある。

#### 4.3.2.4.2. 非同期バッチデーモンのプロセス異常終了

非同期バッチデーモンのプロセスが異常終了した場合は、実行中ジョブのトランザクションは暗黙的にロールバックされる。

ポーリングステータスによる状態はデータベース接続障害と同じになる。

#### 4.3.2.5. DBポーリング処理の停止について

非同期バッチデーモン(**AsyncBatchDeamon**)は、ファイルの生成によって停止する。ファイルが生成されたことを確認後、ポーリング処理を空振りさせ、起動中ジョブの終了を可能な限り待ってから停止する。

#### 4.3.2.6. 非同期実行特有のアプリケーション構成となる点について

非同期実行における特有の構成を説明する。

##### 4.3.2.6.1. ApplicationContextの構成

非同期バッチデーモンは、非同期実行専用の**async-batch-daemon.xml**をApplicationContextとして読み込む。同期実行でも使用している**launch-context.xml**の他に次の構成を追加している。

##### 非同期実行設定

**JobRequestPollTask**などの非同期実行に必要なBeanを定義している。

##### ジョブ登録設定

非同期実行として実行するジョブ

は、**org.springframework.batch.core.configuration.support.AutomaticJobRegistrar**で登録を行う。**AutomaticJobRegistrar**を用いることで、ジョブ単位にコンテキストのモジュール化を行っている。モジュール化することにより、ジョブ間で利用するBeanIDが重複していても問題にならないようしている。



モジュール化とは

モジュール化とは、「共通定義-各ジョブ定義」の階層構造になっており、各ジョブで定義されたBeanは、ジョブ間で独立したコンテキストに属することである。各ジョブ定義で定義されていないBeanへの参照がある場合は、共通定義で定義されたBeanを参照することになる。

#### 4.3.2.6.2. Bean定義の構成

ジョブのBean定義は、同期実行のBean定義と同じ構成でよい。ただし、以下の注意点がある。

- AutomaticJobRegistrarでジョブを登録する際、ジョブのBeanIDは識別子となるため重複をしてはいけない。
- ステップのBeanIDも重複しないことが望ましい。
  - 設計時に、BeanIDの命名規則を{ジョブID}.{ステップID}とすることで、ジョブIDのみ一意に設計すればよい。

ジョブのBean定義におけるjob-base-context.xmlのインポートは、同期実行と非同期実行で挙動が異なる。



- 同期実行では、job-base-context.xmlから更にlaunch-context.xmlをインポートする。
- 非同期実行では、job-base-context.xmlからlaunch-context.xmlをインポートしない。その代わりにAsyncBatchDeamonがロードするasync-batch-daemon.xmlにて、launch-context.xmlをインポートする。

これは、Spring Batchを起動する際に必要な各種Beanは各ジョブごとにインスタンス化する必要はないことに起因する。Spring Batchの起動に必要な各種Beanは各ジョブの親となる共通定義(async-batch-daemon.xml)にて1つだけ生成すればよい。

### 4.3.3. How to use

#### 4.3.3.1. 各種設定

##### 4.3.3.1.1. ポーリング処理の設定

非同期実行に必要な設定は、batch-application.propertiesで行う。

## *batch-application.properties*

```
#(1)
# Admin DataSource settings.
admin.jdbc.driver=org.postgresql.Driver
admin.jdbc.url=jdbc:postgresql://localhost:5432/postgres
admin.jdbc.username=postgres
admin.jdbc.password=postgres

# TERASOLUNA AsyncBatchDaemon settings.
# (2)
async-batch-daemon.schema.scriptclasspath:org/terasoluna/batch/async/db/schema-
postgresql.sql
# (3)
async-batch-daemon.job-concurrency-num=3
# (4)
async-batch-daemon.polling-interval=5000
# (5)
async-batch-daemon.polling-initial-delay=1000
# (6)
async-batch-daemon.polling-stop-file-path=/tmp/end-async-batch-daemon
```

### 設定内容の項目一覧

項目番	説明
(1)	ジョブ要求テーブルが格納されているデータベースへの接続設定。 デフォルトではJobRepositoryの設定を使用する。
(2)	ジョブ要求テーブルを定義するDDLのパス。 非同期バッチデーモン起動時にジョブ要求テーブルがない場合は、自動生成される。 これは主に試験用機能であり、batch-application.properties内の data-source.initialize.enabledで実行可否を設定できる。 詳細な定義はasync-batch-daemon.xml内の<jdbc:initialize-database>を参照のこと。
(3)	ポーリング時に一括で取得する件数の設定。この設定値は同時並行数としても用いる。
(4)	ポーリング周期の設定。単位はミリ秒。
(5)	ポーリング初回起動遅延時間の設定。単位はミリ秒。
(6)	終了ファイルパスの設定。

環境変数による設定値の変更

`batch-application.properties`の設定値は、同名の環境変数を定義することで設定の変更が可能である。

環境変数が設定された場合は、プロパティ値より優先して使用される。

これは、以下のBean定義に起因する。

`launch-context.xml`の設定箇所



```
<context:property-placeholder location="classpath:batch-
application.properties"
    system-properties-mode="OVERRIDE"
    ignore-resource-not-found="false"
    ignore-unresolvable="true"
    order="1"/>
```

詳細については、TERASOLUNA Server 5.x 開発ガイドラインの[プロパティファイル定義方法について](#)を参照。

#### 4.3.3.1.2. ジョブの設定

非同期実行する対象のジョブは、`async-batch-daemon.xml`の`automaticJobRegistrar`に設定する。以下に初期設定を示す。

`async-batch-daemon.xml`

```
<bean id="automaticJobRegistrar"

class="org.springframework.batch.core.configuration.support.AutomaticJobRegistrar">
    <property name="applicationContextFactories">
        <bean
class="org.springframework.batch.core.configuration.support.ClasspathXmlApplicationCon
textsFactoryBean">
            <property name="resources">
                <list>
                    <value>classpath:/META-INF/jobs/**/*.xml</value> <!-- (1) -->
                </list>
            </property>
        </bean>
    </property>
    <property name="jobLoader">
        <bean
class="org.springframework.batch.core.configuration.support.DefaultJobLoader"
            p:jobRegistry-ref="jobRegistry" />
    </property>
</bean>
```

設定内容の項目一覧

項目番	説明
(1)	非同期実行するジョブBean定義のパス。

登録ジョブの絞込みについて

登録するジョブは、非同期実行することを前提に設計・実装されたジョブを指定すること。非同期で実行することを想定していないジョブを含めて指定すると、ジョブ登録時に意図しない参照により例外が発生することもあるので注意すること。

絞込の例

```
<bean id="automaticJobRegistrar"
      class="org.springframework.batch.core.configuration.support.AutomaticJobRegistrar">
    <property name="applicationContextFactories">
      <bean
        class="org.springframework.batch.core.configuration.support.ClasspathXmlApplicationContextsFactoryBean">
        <property name="resources">
          <list>
            <!-- For the async directory and below -->
            <value>classpath:/META-INF/jobs/aysnc/**/*.xml</value>
            <!-- For a specific job -->
            <value>classpath:/META-INF/jobs/CASE100/SpecialJob.xml</value>
          </list>
        </property>
      </bean>
    </property>
    <property name="jobLoader">
      <bean
        class="org.springframework.batch.core.configuration.support.DefaultJobLoader"
        p:jobRegistry-ref="jobRegistry" />
    </property>
  </bean>
```



ジョブパラメータの入力値検証

**JobPollingTask**は、ジョブ要求テーブルから取得したレコードについて妥当性検証をしない。

よって、テーブルに登録する側にてジョブ名やジョブパラメータについて検証することが望ましい。

ジョブ名が誤っていると、ジョブを起動するが見つからず、例外が発生してしまう。

ジョブパラメータが誤っていると、ジョブは起動するが誤動作してしまう。

ジョブパラメータに限っては、ジョブ起動後に検証を行うことができる。ジョブパラメータの検証については、[パラメータの妥当性検証](#)を参照のこと。



### ジョブ設計上の留意点



非同期実行(DBポーリング)の特性上、同一ジョブの並列実行が可能になっているので、並列実行した場合に同一ジョブが影響を与えないようにする必要がある。

#### 4.3.3.2. 非同期処理の起動から終了まで

非同期バッチデーモンの起動と終了、ジョブ要求テーブルへの登録方法について説明する。

##### 4.3.3.2.1. 非同期バッチデーモンの起動

TERASOLUNA Batch 5.xが提供する、[AsyncBatchDaemon](#)を起動する。

*AsyncBatchDaemon*の起動

```
# Start AsyncBatchDaemon
$ java -cp dependency/* org.terasoluna.batch.async.db.AsyncBatchDaemon
```

この場合、[META-INF/spring/async-batch-daemon.xml](#)を読み込み各種Beanを生成する。

また、別途カスタマイズした[async-batch-daemon.xml](#)を利用したい場合は第一引数に指定して[AsyncBatchDaemon](#)を起動することで実現できる。

引数に指定するBean定義ファイルは、クラスパスからの相対パスで指定すること。

なお、第二引数以降は無視される。

カスタマイズした[META-INF/spring/customized-async-batch-daemon.xml](#)を利用する場合

```
# Start AsyncBatchDaemon
$ java -cp dependency/* org.terasoluna.batch.async.db.AsyncBatchDaemon \
    META-INF/spring/customized-async-batch-daemon.xml
```

[async-batch-daemon.xml](#)のカスタマイズは、ごく一部の設定を変更する場合は直接修正してよい。

しかし、大幅な変更を加える場合や、後述する[複数起動](#)にて複数の設定を管理する場合は、別途ファイルを作成して管理するほうが扱いやすい。

ユーザの状況に応じて選択すること。



dependency配下には、実行に必要なjar一式が格納されている前提とする。

#### 4.3.3.2.2. ジョブの要求

INSERT文のSQLを発行することでジョブ要求テーブルに登録を行う。

PostgreSQLの場合

```
INSERT INTO batch_job_request(job_name,job_parameter,polling_status,create_date)
VALUES ('JOB01', 'param1=dummy,param2=100', 'INIT', current_timestamp);
```

#### 4.3.3.2.3. 非同期バッチデーモンの停止

`batch-application.properties`に設定した終了ファイルを置く。

```
$ touch /tmp/end-async-batch-daemon
```

非同期バッチデーモン起動前に終了ファイルがある場合



非同期バッチデーモン起動前に終了ファイルがある場合、非同期バッチデーモンは即時終了する。非同期バッチデーモンは、終了ファイルがない状態で起動する必要がある。

#### 4.3.3.3. ジョブのステータス確認

ジョブの状態管理はSpring Batchから提供される`JobRepository`で行い、ジョブ要求テーブルではジョブのステータスを管理しない。ジョブ要求テーブルでは`job_execution_id`のカラムをもち、このカラムに格納される値により個々の要求に対するジョブのステータスを確認できるようにしている。ここでは、SQLを直接発行してジョブのステータスを確認する簡単な例を示す。ジョブステータス確認の詳細は、[状態の確認](#)を参照のこと。

PostgreSQLの場合

```
SELECT job_execution_id FROM batch_job_request WHERE job_seq_id = 1;  
  
job_execution_id  
-----  
 2  
(1 row)  
  
SELECT * FROM batch_job_execution WHERE job_execution_id = 2;  
  
job_execution_id | version | job_instance_id |      create_time      |  
start_time       |         end_time        | status   | exit_code | exit_message |  
ocation  
-----+-----+-----+-----+-----+-----+  
-----+-----+-----+-----+-----+  
-----+-----+  
-----  
 2 |      2 |          2 | 2017-02-06 20:54:02.263 | 2017-02-06 20:  
:54:02.295 | 2017-02-06 20:54:02.428 | COMPLETED | COMPLETED |  
(1 row)
```

#### 4.3.3.4. ジョブが異常終了した後のリカバリ

異常終了したジョブのリカバリに関する基本事項は、[処理の再実行](#)を参照のこと。ここでは、非同期実行特有の事項について説明をする。

##### 4.3.3.4.1. リラン

異常終了したジョブのリランは、ジョブ要求テーブルに別レコードとしてINSERTすることで行う。

#### 4.3.3.4.2. リスタート

異常終了したジョブをリスタートする場合は、コマンドラインから同期実行ジョブとして実行する。コマンドラインからの実行する理由は、「意図したリスタート実行なのか意図しない重複実行であるかの判断が難しいため、運用で混乱をきたす可能性がある」ためである。

リスタート方法は[ジョブのリスタート](#)を参照のこと。

#### 4.3.3.4.3. 停止

1. 処理時間が想定を超えて停止していない場合は、コマンドラインからの停止を試みる。停止方法は[ジョブの停止](#)を参照のこと。
2. コマンドラインからの停止も受け付けない場合は、[非同期バッチデーモンの停止](#)により、非同期バッチデーモンを終了させる。
3. 非同期バッチデーモンも終了できない状態になっている場合は、非同期バッチデーモンのプロセスを強制終了させる。



非同期バッチデーモンを終了させる場合は、他のジョブに影響がないように十分に注意して行う。

#### 4.3.3.5. 環境配備について

ジョブのビルドとデプロイは同期実行と同じである。ただし、[ジョブの設定](#)にもあるとおり非同期実行するジョブの絞込みをしておくことが重要である。

#### 4.3.3.6. 累積データの退避について

非同期バッチデーモンを長期運用していると[JobRepository](#)とジョブ要求テーブルに膨大なデータが累積されていく。以下の理由によりこれらの累積データを退避させる必要がある。

- 膨大なデータ量に対してデータを検索/更新する際の性能劣化
- IDの採番用シーケンスが周回することによるIDの重複

テーブルデータの退避やシーケンスのリセットについては、利用するデータベースのマニュアルを参照してほしい。

以下に退避対象のテーブルおよびシーケンスの一覧を示す。

##### 退避対象一覧

テーブル/シーケンス	提供しているフレームワーク
batch_job_request	TERASOLUNA Batch 5.x
batch_job_request_seq	

テーブル/シーケンス	提供しているフレームワーク
batch_job_instance	Spring Batch
batch_job_execution	
batch_job_execution_params	
batch_job_execution_context	
batch_step_execution	
batch_step_execution_context	
batch_job_seq	
batch_job_execution_seq	
batch_step_execution_seq	

#### 自動採番カラムのシーケンス



自動採番のカラムに対して自動的にシーケンスが作成されている場合があるので、忘れずにそのシーケンスも退避対象に含める。

#### データベース固有の仕様について



Oracleではデータ型にCLOBを利用するなど、データベース固有のデータ型を使用している場合があるので注意をする。

### 4.3.4. How to extend

#### 4.3.4.1. ジョブ要求テーブルのカスタマイズ

ジョブ要求テーブルは、取得レコードの抽出条件を変更するためにカラム追加をしてカスタマイズすることができる。ただし、[JobRequestPollTask](#)からSQLを発行する際に渡せる項目は、[BatchJobRequest](#)の項目のみである。

ジョブ要求テーブルのカスタマイズによる拡張手順は以下のとおり。

1. ジョブ要求テーブルのカスタマイズ
2. [BatchJobRequestMapper](#)インターフェースの拡張インターフェースの作成
3. カスタマイズしたテーブルを使用したSQLMapの定義
4. [async-batch-daemon.xml](#)のBean定義の修正

カスタマイズ例として以下のようなものがある。

- [優先度カラムによるジョブ実行順序の制御の例](#)
- [グループIDによる複数プロセスによる分散処理](#)

以降、この2つの例について、拡張手順を説明する。

#### 4.3.4.1.1. 優先度カラムによるジョブ実行順序の制御の例

##### 1. ジョブ要求テーブルのカスタマイズ

ジョブ要求テーブルに優先度カラム(priority)を追加する。

優先度カラムの追加 (*PostgreSQL*の場合)

```
CREATE TABLE IF NOT EXISTS batch_job_request (
    job_seq_id bigserial PRIMARY KEY,
    job_name varchar(100) NOT NULL,
    job_parameter varchar(200),
    priority int NOT NULL,
    job_execution_id bigint,
    polling_status varchar(10) NOT NULL,
    create_date timestamp NOT NULL,
    update_date timestamp
);
```

##### 2. `BatchJobRequestMapper`インターフェースの拡張インターフェースの作成

`BatchJobRequestMapper`インターフェースを拡張したインターフェースを作成する。

拡張インターフェース

```
// (1)
public interface CustomizedBatchJobRequestMapper extends BatchJobRequestMapper {
    // (2)
}
```

拡張ポイント

項番	説明
(1)	<code>BatchJobRequestMapper</code> を拡張する。
(2)	メソッドは追加しない。

##### 3. カスタマイズしたテーブルを使用したSQLMapの定義

優先度を順序条件にしたSQLをSQLMapに定義する。

## SQLMap定義(CustomizedBatchJobRequestMapper.xml)

```
<!-- (1) -->
<mapper
namespace="org.terasoluna.batch.extend.repository.CustomizedBatchJobRequestMapper">

    <select id="find" resultType=
"org.terasoluna.batch.async.db.model.BatchJobRequest">
        SELECT
            job_seq_id AS jobSeqId,
            job_name AS jobName,
            job_parameter AS jobParameter,
            job_execution_id AS jobExecutionId,
            polling_status AS pollingStatus,
            create_date AS createDate,
            update_date AS updateDate
        FROM
            batch_job_request
        WHERE
            polling_status = 'INIT'
        ORDER BY
            priority ASC,    <!--(2) -->
            job_seq_id ASC
        LIMIT #{pollingRowLimit}
    </select>

    <!-- (3) -->
    <update id="updateStatus">
        UPDATE
            batch_job_request
        SET
            polling_status = #{batchJobRequest.pollingStatus},
            job_execution_id = #{batchJobRequest.jobExecutionId},
            update_date = #{batchJobRequest.updateDate}
        WHERE
            job_seq_id = #{batchJobRequest.jobSeqId}
        AND
            polling_status = #{pollingStatus}
    </update>

</mapper>
```

### 拡張ポイント

項番	説明
(1)	BatchJobRequestMapperの拡張インターフェースをFQCNでnamespaceに設定する。
(2)	priorityをORDER句へ追加する。
(3)	更新SQLは変更しない。

#### 4. `async-batch-daemon.xml`のBean定義の修正

(2)で作成した拡張インターフェースを`batchJobRequestMapper`に設定する。

`async-batch-daemon.xml`

```
<!--(1) -->
<bean id="batchJobRequestMapper"
      class="org.mybatis.spring.mapper.MapperFactoryBean"
      p:mapperInterface="org.terasoluna.batch.extend.repository.CustomizedBatchJobRequestMapper"
      p:sqlSessionFactory-ref="adminSqlSessionFactory" />
```

#### 拡張ポイント

項目番	説明
(1)	<code>BatchJobRequestMapper</code> の拡張インターフェースをFQCNで <code>mapperInterface</code> プロパティに設定する。

##### 4.3.4.1.2. グループIDによる複数プロセスによる分散処理

`AsyncBatchDaemon`起動時に環境変数でグループIDを指定して、対象のジョブを絞り込む。

###### 1. ジョブ要求テーブルのカスタマイズ

ジョブ要求テーブルにグループIDカラム(`group_id`)を追加する。

グループIDカラムの追加 (*PostgreSQL*の場合)

```
CREATE TABLE IF NOT EXISTS batch_job_request (
    job_seq_id bigserial PRIMARY KEY,
    job_name varchar(100) NOT NULL,
    job_parameter varchar(200),
    group_id varchar(10) NOT NULL,
    job_execution_id bigint,
    polling_status varchar(10) NOT NULL,
    create_date timestamp NOT NULL,
    update_date timestamp
);
```

###### 2. `BatchJobRequestMapper`インターフェースの拡張インターフェース作成

- 優先度カラムによるジョブ実行順序の制御の例と同じ

###### 3. カスタマイズしたテーブルを使用したSQLMapの定義

優先度を順序条件にしたSQLをSQLMapに定義する。

## SQLMap定義(CustomizedBatchJobRequestMapper.xml)

```
<!-- (1) -->
<mapper
namespace="org.terasoluna.batch.extend.repository.CustomizedBatchJobRequestMapper">

    <select id="find" resultType=
"org.terasoluna.batch.async.db.model.BatchJobRequest">
        SELECT
            job_seq_id AS jobSeqId,
            job_name AS jobName,
            job_parameter AS jobParameter,
            job_execution_id AS jobExecutionId,
            polling_status AS pollingStatus,
            create_date AS createDate,
            update_date AS updateDate
        FROM
            batch_job_request
        WHERE
            polling_status = 'INIT'
        AND
            group_id = #{groupId} <!--(2) -->
        ORDER BY
            job_seq_id ASC
        LIMIT #{pollingRowLimit}
    </select>

    <!-- ommited -->
</mapper>
```

### 拡張ポイント

項番	説明
(1)	BatchJobRequestMapperの拡張インターフェースをFQCNでnamespaceに設定する。
(2)	groupIdを検索条件に追加。

### 4. `async-batch-daemon.xml`のBean定義の修正

(2)で作成した拡張インターフェースをbatchJobRequestMapperに設定し、jobRequestPollTaskに環境変数で与えられたグループIDをクエリパラメータとして設定する。

## async-batch-daemon.xml

```
<!--(1) -->
<bean id="batchJobRequestMapper"
      class="org.mybatis.spring.mapper.MapperFactoryBean"
      p:mapperInterface="org.terasoluna.batch.extend.repository.CustomizedBatchJobRequestMapper"
      p:sqlSessionFactory-ref="adminSqlSessionFactory" />

<bean id="jobRequestPollTask"
      class="org.terasoluna.batch.async.db.JobRequestPollTask"
      c:transactionManager-ref="adminTransactionManager"
      c:jobOperator-ref="jobOperator"
      c:batchJobRequestMapper-ref="batchJobRequestMapper"
      c:daemonTaskExecutor-ref="daemonTaskExecutor"
      c:automaticJobRegistrar-ref="automaticJobRegistrar"
      p:optionalPollingQueryParams-ref="pollingQueryParam" /> <!-- (2) -->

<bean id="pollingQueryParam"
      class="org.springframework.beans.factory.config.MapFactoryBean">
    <property name="sourceMap">
      <map>
        <entry key="groupId" value="${GROUP_ID}" /> <!-- (3) -->
      </map>
    </property>
  </bean>
```

## 拡張ポイント

項目番号	説明
(1)	BatchJobRequestMapperの拡張インターフェースをFQCNでmapperInterfaceプロパティに設定する。
(2)	JobRequestPollTaskのoptionalPollingQueryParamsプロパティに(3)で定義するMapを設定する。
(3)	環境変数で与えられたグループID(GROUP_ID)をクエリパラメータのグループID(groupId)に設定する。

5. 環境変数にグループIDを設定後、**AsyncBatchDaemon**を起動する。

## AsyncBatchDaemonの起動

```
# Set environment variables
$ export GROUP_ID=G1

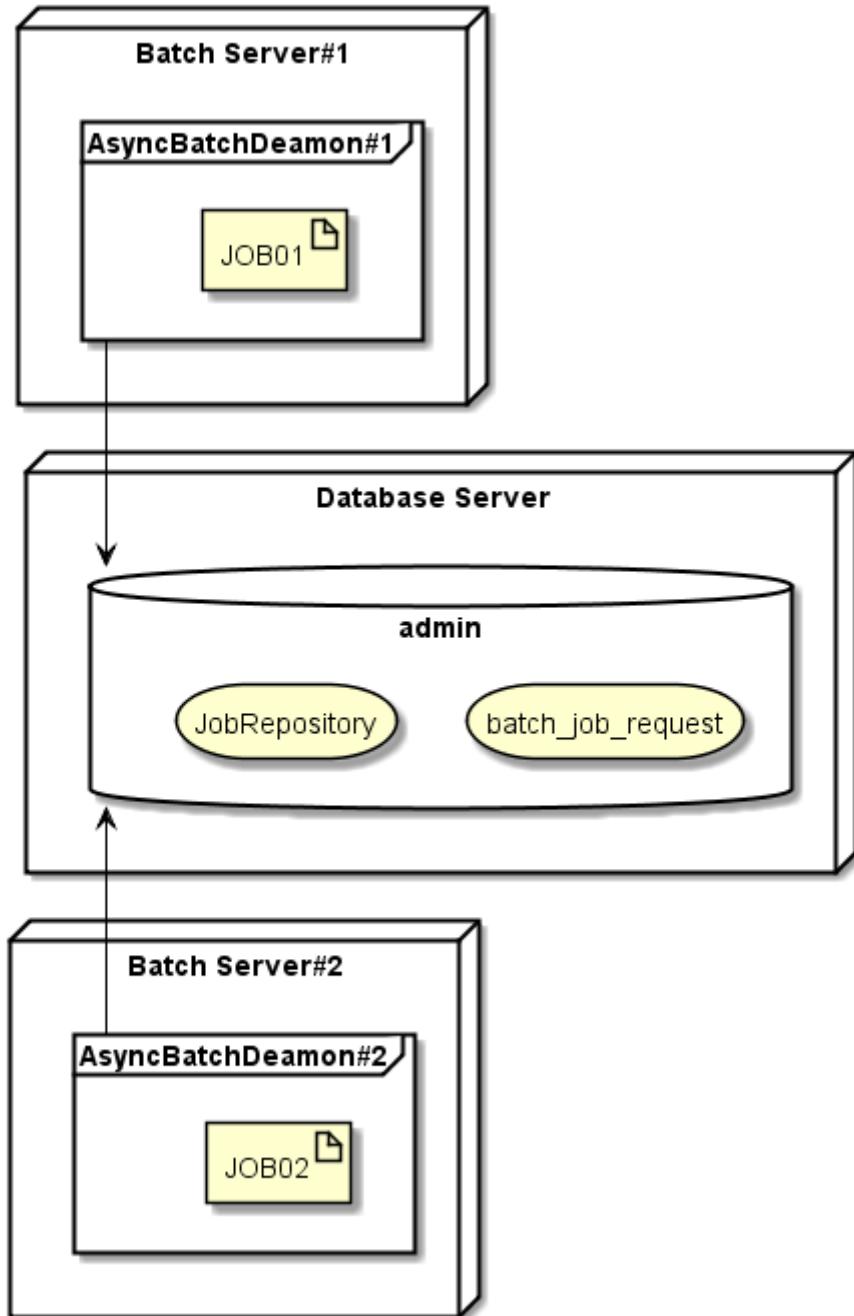
# Start AsyncBatchDaemon
$ java -cp dependency/* org.terasoluna.batch.async.db.AsyncBatchDaemon
```

#### 4.3.4.2. 複数起動

以下の様な目的で、複数サーバ上で非同期バッチデーモンを起動させる場合がある。

- 可用性向上
  - 非同期バッチジョブがいずれかのサーバで実行できればよく、ジョブが起動できないという状況をなくしたい場合
- 性能向上
  - 複数サーバでバッチ処理の負荷を分散させたい場合
- リソースの有効利用
  - サーバ性能に差がある場合に特定のジョブを最適なリソースのサーバに振り分ける場合
    - [ジョブ要求テーブルのカスタマイズ](#)で提示したグループIDによるジョブノードの分割に相当

上記に示す観点のいずれかにもとづいて利用するのかを意識して運用設計を行うことが必要となる。



複数起動の概略図

複数の非同期バッチデーモンが同一ジョブ要求レコードを取得した場合



`JobRequestPollTask`は、楽観ロックによる排他制御を行っているため、ポーリングステータスをINITからPOLLLEDへ更新できた非同期バッチデーモンが取得したレコードのジョブを実行できる。排他された他の非同期バッチデーモンは、次のジョブ要求レコードを処理する。

#### 4.3.5. Appendix

##### 4.3.5.1. ジョブ定義のモジュール化について

[ApplicationContextの構成](#)でも簡単に説明したが、[AutomaticJobRegistrar](#)を用いることで以下の事象を回避することができる。

- 同じBeanID(BeanName)を使用すると、Beanが上書きされてしまい、ジョブが意図しない動作をする。
  - その結果、意図しないエラーが発生する可能性が高くなる。
- エラーを回避するために、ジョブ全体でBeanすべてのIDが一意になるように命名しなければいけなくなる。
  - ジョブ数が増えてくると管理するのが困難になり、不必要的トラブルが発生する可能性が高くなる。

`AutomaticJobRegistrar`を使用しない場合に起こる現象について説明をする。ここで説明する内容は上記の問題を引き起こすので、非同期実行では使用しないこと。

### Job1.xml

```

<!-- Reader -->
<!-- (1) -->
<bean id="reader" class="org.mybatis.spring.batch.MyBatisCursorItemReader"
      p:queryId="jp.terasoluna.batch.job.repository.EmployeeRepository.findAll"
      p:sqlSessionFactory-ref="jobSqlSessionFactory"/>

<!-- Writer -->
<!-- (2) -->
<bean id="writer"
      class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step"
      p:resource="file:${jobParameters[basedir]}/input/employee.csv">
  <property name="lineAggregator">
    <bean
      class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
      <property name="fieldExtractor">
        <bean
          class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"
          p:names="invoiceNo,salesDate,productId,customerId,quant,price"/>
        </property>
      </bean>
    </property>
  </bean>
</bean>

<!-- Job -->
<batch:job id="job1" job-repository="jobRepository">
  <batch:step id="job1.step">
    <batch:tasklet transaction-manager="transactionManager">
      <batch:chunk reader="reader" writer="writer" commit-interval="100" />
    </batch:tasklet>
  </batch:step>
</batch:job>
```

## Job2.xml

```
<!-- Reader -->
<!-- (3) -->
<bean id="reader"
    class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
    p:resource="file:#{jobParameters[basedir]}/input/invoice.csv">
    <property name="lineMapper">
        <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <property name="lineTokenizer">
                <bean
                    class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
                    p:names="invoiceNo,salesDate,productId,customerId,quant,price"/>
            </property>
            <property name="fieldSetMapper" ref="invoiceFieldSetMapper"/>
        </bean>
    </property>
</bean>

<!-- Writer -->
<!-- (4) -->
<bean id="writer" class="org.mybatis.spring.batch.MyBatisBatchItemWriter"
    p:statementId="jp.terasoluna.batch.job.repository.InvoiceRepository.create"
    p:sqlSessionFactory-ref="jobSqlSessionFactory"/>

<!-- Job -->
<batch:job id="job2" job-repository="jobRepository">
    <batch:step id="job2.step">
        <batch:tasklet transaction-manager="transactionManager">
            <batch:chunk reader="reader" writer="writer" commit-interval="100" />
        </batch:tasklet>
    </batch:step>
</batch:job>
```

## BeanIdが上書きされる定義

```
<bean id="automaticJobRegistrar"
      class="org.springframework.batch.core.configuration.support.AutomaticJobRegistrar">
    <property name="applicationContextFactories">
      <bean
        class="org.springframework.batch.core.configuration.support.ClasspathXmlApplicationCon-
tentsFactoryBean">
        <property name="resources">
          <list>
            <value>classpath:/META-INF/jobs/other/async/*.xml</value>  <!--
(5) -->
          </list>
        </property>
      </bean>
    </property>
    <property name="jobLoader">
      <bean
        class="org.springframework.batch.core.configuration.support.DefaultJobLoader"
        p:jobRegistry-ref="jobRegistry"/>
    </property>
  </bean>

  <bean
    class="org.springframework.batch.core.configuration.support.JobRegistryBeanPostProcess-
or"
    p:jobRegistry-ref="jobRegistry" />

  <import resource="classpath:/META-INF/jobs/async/*.xml" />  <!-- (6) -->
```

## 設定のポイント一覧

項目番号	説明
(1)	Job1ではデータベースから読み込むItemReaderをreaderというBeanIDで定義する。
(2)	Job1ではファイルへ書き込むItemWriterをwriterというBeanIDで定義する。
(3)	Job2ではファイルから読み込むItemReaderをreaderというBeanIDで定義する。
(4)	Job2ではデータベースへ書き込むItemWriterをwriterというBeanIDで定義する。
(5)	AutomaticJobRegistrarは対象となるジョブ以外のジョブ定義を読む込むように設定する。
(6)	Springのimportを使用して、対象のJob定義を読み込むようにする。

この場合、Job1.xml,Job2.xmlの順に読み込まれたとすると、Job1.xmlで定義されたいたreader,writerはJob2.xmlの定義で上書きされる。

その結果、Job1を実行すると、Job2のreader,writerが使用されて期待した処理が行われなくなる。

## 4.4. 非同期実行(Webコンテナ)

### 4.4.1. Overview

Webコンテナ内でジョブを非同期で実行するための方法について説明する。

本機能は、チャンクモデルとタスクレットモデルとで同じ使い方になる。

Webコンテナによるジョブの非同期実行とは

ジョブを含めたWebアプリケーションをWebコンテナにデプロイし、送信されたリクエストの情報を元にジョブを実行することを指す。

ジョブの実行ごとに1つのスレッドを割り当てた上で並列に動作するため、他のジョブやリクエストに対する処理とは独立して実行できる。

#### 提供機能

TERASOLUNA Batch 5.xでは、非同期実行(Webコンテナ)向けの実装は提供しない。

本ガイドラインにて実現方法を提示するのみとする。

これは、Webアプリケーションの起動契機はHTTP/SOAP/MQなど多様であるため、ユーザにて実装することが適切と判断したためである。

#### 利用前提

- ・ アプリケーションの他にWebコンテナが必要となる。
- ・ ジョブの実装以外に必要となる、Webアプリケーション、クライアントは動作要件に合わせて別途実装する。
- ・ ジョブの実行状況および結果はJobRepositoryに委ねる。また、Webコンテナ停止後にもJobRepositoryからジョブの実行状況および結果を参照可能とするため、インメモリデータベースではなく、永続性が担保されているデータベースを使用する。

#### 活用シーン

非同期実行(DBポーリング) - Overviewと同様である。

##### 非同期実行(DBポーリング)との違い

アーキテクチャ上、非同期実行時の即時性と、要求管理テーブルの有無、の2点が異なる。

非同期実行(DBポーリング)は要求管理テーブルに登録された複数のジョブが一定の周期で非同期実行される。

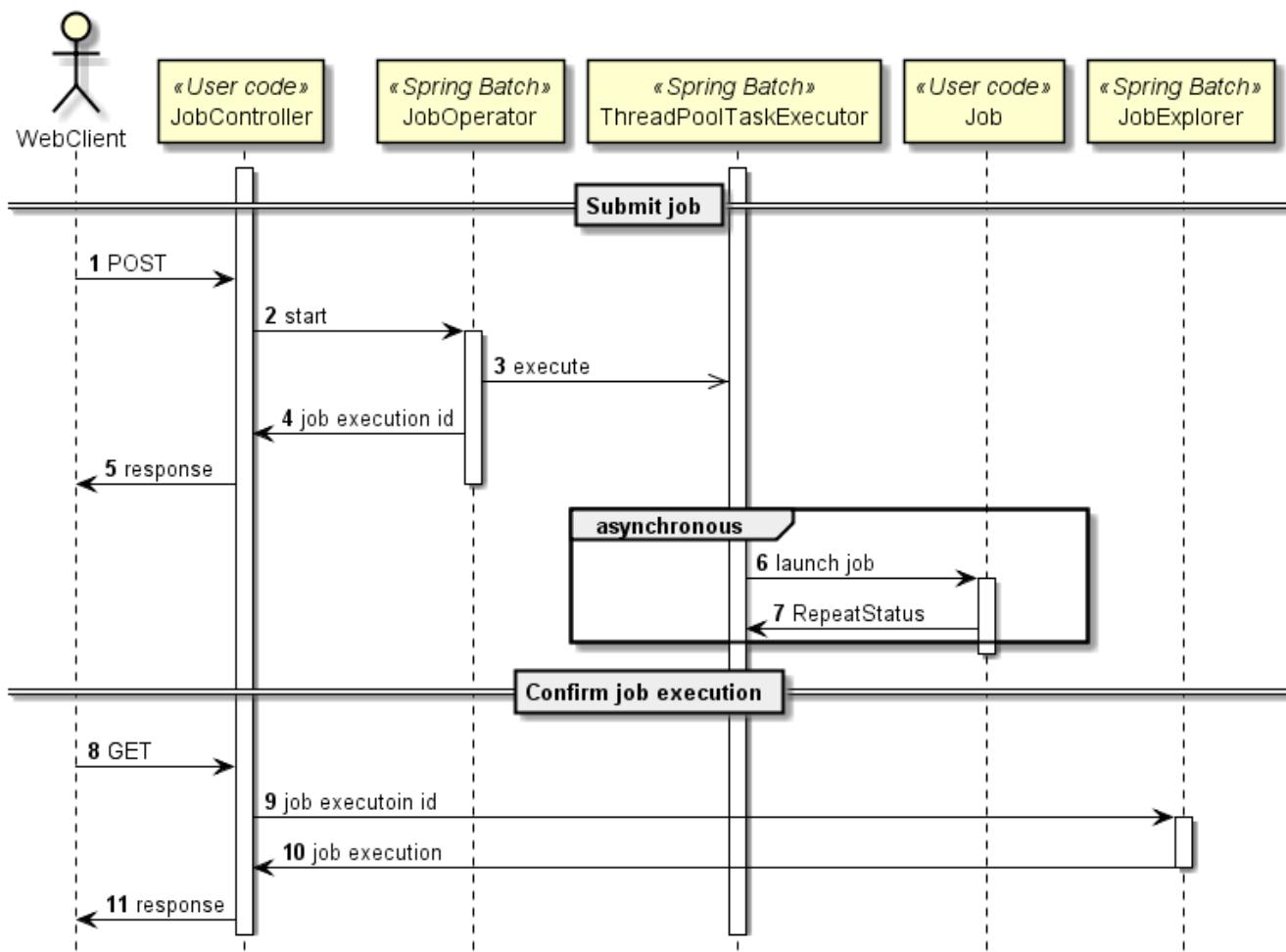
それに対し、本機能は要求管理テーブルを必要とせず代わりにWebコンテナ上で非同期実行を受け付ける。

Webリクエスト送信により直ちに実行するため、起動までの即時性が求められるショートバッチに向いている。



### 4.4.2. Architecture

本方式による非同期ジョブはWebコンテナ上にデプロイされたアプリケーション(war)として動作するが、ジョブ自身はWebコンテナのリクエスト処理とは非同期(別スレッド)で動作する。



非同期実行(Webコンテナ)の処理シーケンス図

### ジョブの起動

1. Webクライアントは実行対象のジョブをWebコンテナに要求する。
2. **JobController**はSpring Batchの**JobOperator**に対しジョブの実行開始を依頼する。
3. **ThreadPoolTaskExecutor**によって非同期でジョブを実行する。
4. 実行された対象のジョブを一意に判別するためのジョブ実行ID(**job execution id**)を返却する。
5. **JobController**はWebクライアントに対し、ジョブ実行IDを含むレスポンスを返却する。
6. 目的のジョブを実行する。
  - ・ ジョブの結果は**JobRepository**に反映される。
7. **Job**が実行結果を返却する。これはクライアントへ直接通知できない。

### ジョブの実行結果確認

8. Webクライアントはジョブ実行IDを**JobController**をWebコンテナに送信する。
9. **JobController**はジョブ実行IDを用い**JobExplorer**にジョブの実行結果を問い合わせる。
10. **JobExplorer**はジョブの実行結果を返却する。
11. **JobController**はWebクライアントに対しレスポンスを返却する。
  - ・ レスポンスにはジョブ実行IDを設定する。

Webコンテナによるリクエスト受信後、ジョブ実行ID払い出しまでがリクエスト処理と同期するが、以降のジョブ実行はWebコンテナとは別のスレッドプールで非同期に行われる。

これは再度リクエストで問い合わせを受けない限り、Webクライアント側では非同期ジョブの実行状態が検知できないことを意味する。

このためWebクライアント側では1回のジョブ実行で、リクエストを「ジョブの起動」で1回、「結果の確認」が必要な場合は加えてもう1回、Webコンテナにリクエストを送信する必要がある。

特に初回の「ジョブの起動」時に見え方が異なる異常検知については、後述の[ジョブ起動時における異常発生の検知について](#)で説明する。



[JobRepository](#)、[JobExplorer](#)を使用して直接RDBMSを参照し、ジョブの実行状態を確認することもできる。ジョブの実行状態・結果を参照する機能の詳細については、[ジョブの管理](#)を参照のこと。



ジョブ実行ID(*job execution id*)の取り扱いについて

ジョブ実行IDは起動対象が同じジョブ、同じジョブパラメータであっても、ジョブ起動ごとに異なるシーケンス値が払い出される。

リクエスト送信により受付が行われたジョブ実行IDは[JobRepository](#)により外部RDBMSで永続化される。

しかし、Webクライアントの障害などによりこのIDが消失した場合、ジョブ実行状況の特定・追跡が困難となる。

このため、Webクライアント側ではレスポンスとして返却されたジョブ実行IDをログに記録するなど、ジョブ実行IDの消失に備えておくこと。

#### 4.4.2.1. ジョブ起動時における異常発生の検知について

Webクライアントからジョブの起動リクエストを送信後、ジョブ実行ID払い出しを境にして異常検知の見え方が異なる。

- ジョブ起動時のレスポンスにて異常がすぐ検知できるもの
  - 起動対象のジョブが存在しない。
  - ジョブパラメータの形式誤り。
- ジョブ起動後、Webコンテナに対しジョブ実行状態・結果の問い合わせが必要となるもの
  - ジョブの実行ステータス
  - 非同期ジョブ実行で使用されるスレッドプールが枯渇したことによるジョブの起動失敗



「ジョブ起動時の異常」はSpring MVCコントローラ内で発生する例外として検知できる。ここでは説明を割愛するので、別途TERASOLUNA Server 5.x開発ガイドラインの[例外のハンドリングの実装](#)を参照のこと。

また、ジョブパラメータとして利用するリクエストの入力チェックは必要に応じてSpring MVCのコントローラ内で行うこと。  
具体的な実装方法については、TERASOLUNA Server 5.x開発ガイドラインの[入力チェック](#)を参照のこと。

スレッドプール枯渇によるジョブの起動失敗はジョブ起動時に補足できない  
スレッドプール枯渇によるジョブの起動失敗は、[JobOperator](#)から例外があがってこないため、別途確認する必要がある。確認方法の1つは、ジョブの実行状態確認時に[JobExplorer](#)を用い、以下の条件に合致しているかどうかである。



- ステータスが[FAILED](#)である
- `jobExecution.getExitStatus().getExitDescription()`にて、`org.springframework.core.task.TaskRejectedException`の例外スタックトレースが記録されている

#### 4.4.2.2. 非同期実行(Webコンテナ)のアプリケーション構成

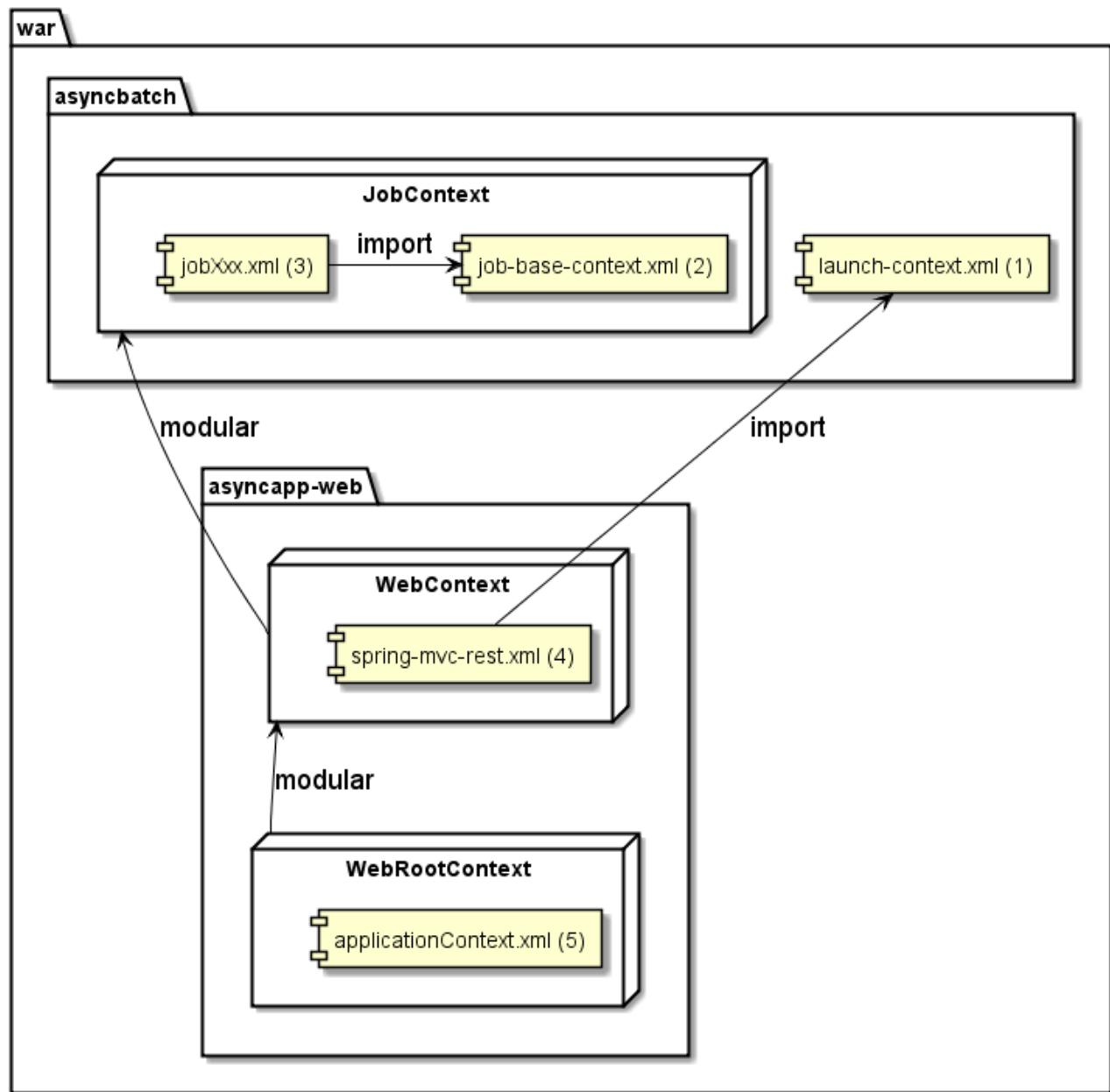
本機能は[非同期実行\(DBポーリング\)](#)と同様、非同期実行特有の構成としてSpring プロファイルの[async](#)と[AutomaticJobRegistrar](#)を使用している。

一方で、これら機能を非同期実行(Webコンテナ)使用する上で、いくつかの事前知識と設定が必要となる。[ApplicationContextの構成](#)を参照。

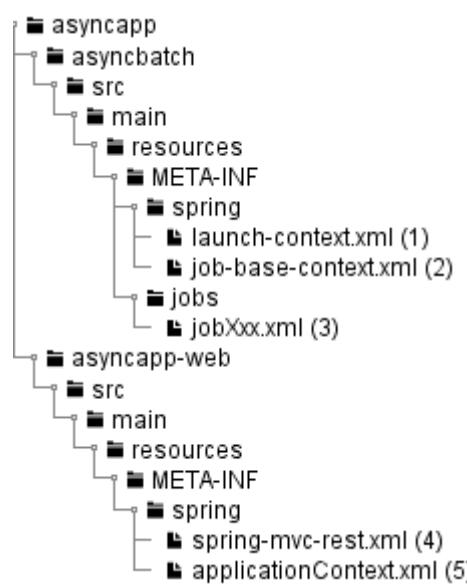
具体的な[async](#)プロファイルと[AutomaticJobRegistrar](#)の設定方法については[非同期実行\(Webコンテナ\)によるアプリケーションの実装方法](#)についてで後述する。

##### 4.4.2.2.1. ApplicationContextの構成

上述のとおり、非同期実行(Webコンテナ)のアプリケーション構成として、複数のアプリケーションモジュールが含まれている。  
それぞれのアプリケーションコンテキストとBean定義についての種類、および関係性を把握しておく必要がある。



ApplicationContextの構成



## Bean定義ファイルの構成

非同期実行(Webコンテナ)におけるApplicationContextでは、バッチャアプリケーションのApplicationContextはWebのコンテキスト内に取り込まれる。

個々のジョブコンテキストはこのWebコンテキストからAutomaticJobRegistrarによりモジュール化され、Webコンテキストの子コンテキストとして動作する。

以下、それぞれのコンテキストを構成するBean定義ファイルについて説明する。

### Bean定義ファイル一覧

項番	説明
(1)	共通Bean定義ファイル。 アプリケーション内では親コンテキストとなり、子コンテキストであるジョブ間で一意に共有される。
(2)	ジョブBean定義から必ずインポートされるBean定義ファイル。 Spring プロファイルが非同期実行時に指定されるasyncの場合は(1)のlaunch-context.xmlを読み込まない。
(3)	ジョブごとに作成するBean定義ファイル。 AutomaticJobRegistrarによりモジュラー化され、アプリケーション内ではそれぞれ独立した子コンテキストとして使用される。
(4)	DispatcherServletから読み込まれる。 ジョブBean定義のモジュラー化を行うAutomaticJobRegistrarや、ジョブの非同期・並列実行で使用されるスレッドプールであるtaskExecutorなど、非同期実行特有のBeanを定義する。 また、非同期実行では(1)のlaunch-context.xmlを直接インポートし親コンテキストとして一意に共有化される。
(5)	ContextLoaderListenerにより、Webアプリケーション内で共有される親コンテキストとなる。

### 4.4.3. How to use

ここでは、Webアプリケーション側の実装例として、TERASOLUNA Server Framework for Java (5.x)を用いて説明する。

あくまで説明のためであり、TERASOLUNA Server 5.xは非同期実行(Webコンテナ)の必須要件ではないことに留意してほしい。

#### 4.4.3.1. 非同期実行(Webコンテナ)によるアプリケーションの実装概要

以下の構成を前提とし説明する。

- Webアプリケーションプロジェクトとバッチャアプリケーションプロジェクトは独立し、webアプリケーションからバッチャアプリケーションを参照する。
- Webアプリケーションプロジェクトから生成するwarファイルは、バッチャアプリケーションプロジェクトから生成されるjarファイルを含むこととなる

非同期実行の実装はArchitectureに従い、Webアプリケーション内のSpring MVCコントローラが、JobOperatorによりジョブを起動する。



### Web/バッチアプリケーションプロジェクトの分離について

アプリケーションビルトの最終成果物はWebアプリケーションのwarファイルであるが、開発プロジェクトはWeb/バッチアプリケーションで分離して実装を行うとよい。これはバッチアプリケーション単体で動作可能なライブラリとなるため、開発プロジェクト上の試験を容易にする他、作業境界とライブラリ依存関係を明確にする効果がある。

以降、Web/バッチの開発について、以下2つを利用する前提で説明する。

- TERASOLUNA Batch 5.xによるバッチアプリケーションプロジェクト
- TERASOLUNA Server 5.xによるWebアプリケーションプロジェクト

バッチアプリケーションプロジェクトの作成および具体的なジョブの実装方法については、[プロジェクトの作成](#)、[タスクレットモデルジョブの作成](#)、[チャンクモデルジョブの作成](#)を参照のこと。ここでは、Webアプリケーションからバッチアプリケーションを起動することに終始する。

ここでは Maven archetype:generate を用い、以下のバッチアプリケーションプロジェクトを作成しているものとして説明する。

#### ジョブプロジェクト作成例

名称	値
groupId	org.terasoluna.batch.sample
archetypeId	asyncbatch
version	1.0-SNAPSHOT
package	org.terasoluna.batch.sample

また説明の都合上、ブランクプロジェクトに初めから登録されているジョブを使用する。

#### 説明に用いるジョブ

名称	説明
ジョブ名	job01
ジョブパラメータ	param1=value1



#### 非同期実行(Webコンテナ)ジョブ設計の注意点

非同期実行(Webコンテナ)の特性として個々のジョブは短時間で完了しWebコンテナ上でステートレスに動作するケースが適している。

また複雑さを避ける上では、ジョブ定義を単一のステップのみで構成し、ステップの終了コードによるフローの分岐や並列処理・多重処理を定義しないことが望ましい。

ジョブ実装を含むjarファイルが作成可能な状態として、Webアプリケーションの作成を行う。

#### Webアプリケーションの実装

TERASOLUNA Server 5.xが提供するブランクプロジェクトを用い、Webアプリケーションの実装方法を説明する。詳細は、TERASOLUNA Server 5.x 開発ガイドライン [Webアプリケーション向け開発プロジ](#)

[エクトの作成](#) を参照。

ここでは非同期実行アプリケーションプロジェクトと同様、以下の名称で作成したものとして説明する。

#### Webコンテナプロジェクト作成例

名称	値
groupId	org.terasoluna.batch.sample
archetypeId	asyncapp
version	1.0-SNAPSHOT
package	org.terasoluna.batch.sample

##### groupIdの命名について



プロジェクトの命名は任意であるが、Maven マルチプロジェクトとしてバッチアプリケーションを Web アプリケーションの子モジュールとする場合、**groupId** は統一しておくと管理しやすい。

ここでは両者の **groupId** を `org.terasoluna.batch.sample` としている。

#### 4.4.3.2. 各種設定

バッチアプリケーションを Web アプリケーションの一部に含める

pom.xml を編集し、バッチアプリケーションを Web アプリケーションの一部に含める。



バッチアプリケーションを **jar** として NEXUS や Maven ローカルリポジトリに登録し、Web アプリケーションとは別プロジェクトとする場合はこの手順は不要である。

ただし、Maven によりビルドされる対象が別プロジェクトとなり、バッチアプリケーションの修正を行っても Web アプリケーションのビルド時に反映されないため注意すること。

バッチアプリケーションの修正を Web アプリケーションに反映させるためには同リポジトリに登録する必要がある。

```
└─■ asyncapp
   └─■ asyncbatch
   └─■ asyncapp-domain
   └─■ asyncapp-env
   └─■ asyncapp-initdb
   └─■ asyncapp-selenium
   └─■ asyncapp-web
```

ディレクトリ構成

### asyncapp/pom.xml

```
<project>
  <!-- omitted -->
  <modules>
    <module>asyncapp-domain</module>
    <module>asyncapp-env</module>
    <module>asyncapp-initdb</module>
    <module>asyncapp-web</module>
    <module>asyncapp-selenium</module>
    <module>asyncbatch</module> <!-- (1) -->
  </modules>
</project>
```

### asyncapp/asyncbatch/pom.xml

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.terasoluna.batch.sample</groupId> <!-- (2) -->
  <artifactId>asyncbatch</artifactId>
  <version>1.0-SNAPSHOT</version> <!-- (2) -->
  <!-- (1) -->
  <parent>
    <groupId>org.terasoluna.batch.sample</groupId>
    <artifactId>asyncapp</artifactId>
    <version>1.0-SNAPSHOT</version>
    <relativePath>../pom.xml</relativePath>
  </parent>
  <!-- omitted -->
</project>
```

### 削除・追加内容

項番	説明
(1)	Webアプリケーションを親とし、バッチアプリケーションを子とするための設定を追記する。
(2)	子モジュール化にともない、不要となる記述を削除する。

### 依存ライブラリの追加

バッチアプリケーションをWebアプリケーションの依存ライブラリとして追加する。

```

<project>
  <!-- omitted -->
  <dependencies>
    <!-- (1) -->
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>asyncbatch</artifactId>
      <version>${project.version}</version>
    </dependency>
    <!-- omitted -->
  </dependencies>
  <!-- omitted -->
</project>

```

## 追加内容

項目番	説明
(1)	バッチアプリケーションをWebアプリケーションの依存ライブラリとして追加する。

### 4.4.3.3. Webアプリケーションの実装

ここではWebアプリケーションとして、以下TERASOLUNA Server 5.x 開発ガイドラインを参考に、RESTful Webサービスを作成する。

RESTful Web Serviceで必要となるSpring MVCのコンポーネントを有効化するための設定

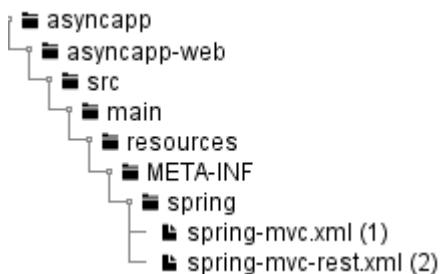
#### 4.4.3.3.1. Webアプリケーションの設定

まず、Webアプリケーションのブランクプロジェクトから、各種設定ファイルの追加・削除・編集を行う。



説明の都合上、バッチアプリケーションの実装形態としてRESTful Web Service を用いた実装を行っている。

従来のWebアプリケーション(Servlet/JSP)やSOAPを使用した場合でも同様な手順となるので、適宜読み替えること。



ブランクプロジェクトから追加・削除するBean定義ファイル

asyncapp/asyncapp-web/src/main/resources/META-INF/spring/spring-mvc-rest.xml の記述例

```
<!-- omitted -->
```

```

<!-- (1) -->
<import resource="classpath:META-INF/spring/launch-context.xml"/>

<bean id="jsonMessageConverter"
      class="org.springframework.http.converter.json.MappingJackson2HttpMessageConverter"
      p:objectMapper-ref="objectMapper"/>

<bean id="objectMapper"
      class="org.springframework.http.converter.json.Jackson2ObjectMapperFactoryBean">
    <property name="dateFormat">
      <bean class="com.fasterxml.jackson.databind.util.StdDateFormat"/>
    </property>
</bean>

<mvc:annotation-driven>
  <mvc:message-converters register-defaults="false">
    <ref bean="jsonMessageConverter"/>
  </mvc:message-converters>
</mvc:annotation-driven>

<mvc:default-servlet-handler/>

<!-- (2) -->
<context:component-scan base-package="org.terasoluna.batch.sample.app.api"/>

<!-- (3) -->
<bean
  class="org.springframework.batch.core.configuration.support.AutomaticJobRegistrar">
  <property name="applicationContextFactories">
    <bean
      class="org.springframework.batch.core.configuration.support.ClasspathXmlApplicationCon
tentsFactoryBean">
      <property name="resources">
        <list>
          <value>classpath:/META-INF/jobs/**/*.xml</value>
        </list>
      </property>
    </bean>
  </property>
  <property name="jobLoader">
    <bean
      class="org.springframework.batch.core.configuration.support.DefaultJobLoader"
      p:jobRegistry-ref="jobRegistry"/>
  </property>
</bean>

<!-- (4) -->
<task:executor id="taskExecutor" pool-size="3" queue-capacity="10"/>

<!-- (5) -->

```

```

<bean id="jobLauncher"
  class="org.springframework.batch.core.launch.support.SimpleJobLauncher"
    p:jobRepository-ref="jobRepository"
    p:taskExecutor-ref="taskExecutor"/>
<!-- omitted -->

```

asyncapp/asyncapp-web/src/main/webapp/WEB-INF/web.xml の記述例

```

<!-- omitted -->
<servlet>
  <servlet-name>restApiServlet</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <!-- (6) -->
    <param-value>classpath*:META-INF/spring/spring-mvc-rest.xml</param-value>
  </init-param>
  <!-- (7) -->
  <init-param>
    <param-name>spring.profiles.active</param-name>
    <param-value>async</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>restApiServlet</servlet-name>
  <url-pattern>/api/v1/*</url-pattern>
</servlet-mapping>
<!-- omitted -->

```

RESTful Web Service の有効化例

項番	説明
(1)	バッチアプリケーション内にある <b>launch-context.xml</b> のimportし、必須となるBean定義を取り込む。
(2)	コントローラを動的にスキャンするためのパッケージを記述する。
(3)	個々のジョブBean定義ファイルをモジュラー化することにより子コンテキストとして動的コードを行う <b>AutomaticJobRegistrar</b> のBean定義を記述する。

項目番	説明
(4)	<p>非同期で実行するジョブが用いる<b>TaskExecutor</b>を定義する。</p> <p>JobLauncherの<b>TaskExecutor</b>に<b>AsyncTaskExecutor</b>実装クラスを設定することで非同期実行が可能となる。 <b>AsyncTaskExecutor</b>実装クラスの1つである<b>ThreadPoolTaskExecutor</b>を利用する。</p> <p>また、並列起動可能なスレッドの多重度を指定ことができる。</p> <p>この例では3スレッドがジョブの実行に割り当てられ、それを超えたリクエストは10までがキューイングされる。 キューイングされたジョブは未開始の状態ではあるが、RESTリクエストは成功とみなされる。</p> <p>さらにキューイングの上限を超えたジョブのリクエストは<code>org.springframework.core.task.TaskRejectedException</code>が発生し、ジョブの起動要求が拒否される。</p>
(5)	(4)の <b>taskExecutor</b> を有効化するため、 <code>launch-context.xml</code> で定義されている <b>jobLauncher</b> をオーバーライドする。
(6)	<b>DispatcherServlet</b> が読み込むBean定義として、上述で記載した <code>spring-mvc-rest.xml</code> を指定する。
(7)	Spring Framework のプロファイルとして、非同期バッチを表す <b>async</b> を明示する。

#### asyncプロファイルの指定をしなかった場合



この場合、Webアプリケーション横断で共有すればよい`launch-context.xml`に定義されたBeanが、ジョブごとに重複して生成される。

重複した場合でも機能上動作するため誤りに気づきにくく、予期しないリソース枯渀や性能劣化が発生する恐れがある。必ず指定すること。

#### スレッドプールのサイジング



スレッドプールの上限が過剰である場合、膨大なジョブが並走することとなり、アプリケーション全体のスループットが劣化する恐れがある。サイジングを行ったうえで適正な上限値を定めること。

非同期実行のスレッドプールとは別に、Webコンテナのリクエストスレッドや同一筐体内で動作している他のアプリケーションも含めて検討する必要がある。

また、スレッドプール枯渀に伴う**TaskRejectException**発生の確認、および再実行はWebクライアントから別途リクエストを送信する必要がある。そのため、スレッドプール枯渀時、ジョブ起動を待機させる**queue-capacity**は必ず設定すること。

### RESTful Web Service API の定義

REST APIで使用するリクエストの例として、ここでは「ジョブの起動」、「ジョブの状態確認」の2つを定義する。

#### REST API 定義例

項目番	API	パス	HTTPメソッド	要求／応答	電文形式	電文の説明
(1)	ジョブの起動	<code>/api/v1/job/ジョブ名</code>	POST	リクエスト	JSON	ジョブパラメータ
				レスポンス	JSON	ジョブ実行ID ジョブ名 メッセージ

項目番	API	パス	HTTPメソッド	要求／応答	電文形式	電文の説明
(2)	ジョブの実行状態確認	/api/v1/job/ジョブ実行ID	GET	リクエスト	N/A	N/A
				レスポンス	JSON	ジョブ実行ID ジョブ名 ジョブ実行ステータス ジョブ終了コード ステップ実行ID ステップ名 ステップ終了コード

#### 4.4.3.3.2. コントローラで使用するJavaBeansの実装

JSON電文としてRESTクライアントに返却される以下3クラスを作成する。

- ジョブ起動操作 `JobOperationResource`
- ジョブの実行状態 `JobExecutionResource`
- ステップの実行状態 `StepExecutionResource`

これらクラスは`JobOperationResource`のジョブ実行ID(`job execution id`)を除きあくまで参考実装であり、フィールドの実装は任意である。

##### ジョブ起動操作情報実装例

```
// asyncapp/asyncapp-
web/src/main/java/org/terasoluna/batch/sample/app/api/jobinfo/JobOperationResource.java
a
package org.terasoluna.batch.sample.app.api.jobinfo;

public class JobOperationResource {

    private String jobName = null;

    private String jobParams = null;

    private Long jobExecutionId = null;

    private String errorMessage = null;

    private Exception error = null;

    // Getter and setter are omitted.
}
```

## ジョブ実行情報実装例

```
// asyncapp/asyncapp-
web/src/main/java/org/terasoluna/batch/sample/app/api/jobinfo/JobExecutionResource.jav
a
package org.terasoluna.batch.sample.app.api.jobinfo;

// omitted.

public class JobExecutionResource {

    private Long jobExecutionId = null;

    private String jobName = null;

    private Long stepExecutionId = null;

    private String stepName = null;

    private List<StepExecutionResource> stepExecutions = new ArrayList<>();

    private String status = null;

    private String exitStatus = null;

    private String errorMessage;

    private List<String> failureExceptions = new ArrayList<>();

    // Getter and setter are omitted.
}
```

## ステップ実行情報実装例

```
// asyncapp/asyncapp-
web/src/main/java/org/terasoluna/batch/sample/app/api/jobinfo/StepExecutionResource.java
va
package org.terasoluna.batch.sample.app.api.jobinfo;

public class StepExecutionResource {

    private Long stepExecutionId = null;

    private String stepName = null;

    private String status = null;

    private List<String> failureExceptions = new ArrayList<>();

    // Getter and setter are omitted.
}
```

### 4.4.3.3. コントローラの実装

@RestControllerを用い、RESTful Web Service のコントローラを実装する。

ここでは簡単のため、JobOperatorをコントローラにインジェクションし、ジョブの起動や実行状態の取得を行う。もちろんTERASOLUNA Server 5.xに従って、コントローラからServiceをはさんでJobOperatorを起動してもよい。



ジョブ起動時に渡されるジョブパラメータについて

起動時にJobOperator#start()の第二引数で渡されるジョブパラメータはStringである。ジョブパラメータが複数ある場合、同期実行のCommandLineJobRunnerとは異なり、カンマ区切りで渡す必要がある。具体的には以下の形式をとる。  
{ジョブパラメータ1}={値1},{ジョブパラメータ2}={値2},…

これは、[非同期実行\(DBポーリング\)](#)におけるジョブパラメータの指定方法と同様である。

## コントローラ実装例

```
// asyncapp/asyncapp-
web/src/main/java/org/terasoluna/batch/sample/app/api/JobController.java
package org.terasoluna.batch.sample.app.api;

// omitted

// (1)
@RequestMapping("job")
@RestController
public class JobController {

    // (2)
```

```

@Inject
JobOperator jobOperator;

// (2)
@Inject
JobExplorer jobExplorer;

@RequestMapping(value = "{jobName}", method = RequestMethod.POST)
public ResponseEntity<JobOperationResource> launch(@PathVariable("jobName") String
jobName,
    @RequestBody JobOperationResource requestResource) {

    JobOperationResource responseResource = new JobOperationResource();
    responseResource.setJobName(jobName);
    try {
        // (3)
        Long jobExecutionId = jobOperator.start(jobName, requestResource
.getJobParams());
        responseResource.setJobExecutionId(jobExecutionId);
        return ResponseEntity.ok().body(responseResource);
    } catch (NoSuchJobException | JobInstanceAlreadyExistsException |
JobParametersInvalidException e) {
        responseResource.setError(e);
        return ResponseEntity.badRequest().body(responseResource);
    }
}

@RequestMapping(value = "{jobExecutionId}", method = RequestMethod.GET)
@ResponseStatus(HttpStatus.OK)
public JobExecutionResource getJob(@PathVariable("jobExecutionId") Long
jobExecutionId) {

    JobExecutionResource responseResource = new JobExecutionResource();
    responseResource.setJobExecutionId(jobExecutionId);

    // (4)
    JobExecution jobExecution = jobExplorer.getJobExecution(jobExecutionId);

    if (jobExecution == null) {
        responseResource.setErrorMessage("Job execution not found.");
    } else {
        mappingExecutionInfo(jobExecution, responseResource);
    }

    return responseResource;
}

private void mappingExecutionInfo(JobExecution src, JobExecutionResource dest) {
    dest.setJobName(src.getJobInstance().getJobName());
    for (StepExecution se : src.getStepExecutions()) {
        StepExecutionResource ser = new StepExecutionResource();

```

```

        ser.setStepExecutionId(se.getId());
        ser.setStepName(se.getStepName());
        ser.setStatus(se.getStatus().toString());
        for (Throwable th : se.getFailureExceptions()) {
            ser.getFailureExceptions().add(th.toString());
        }
        dest.getStepExecutions().add(ser);
    }
    dest.setStatus(src.getStatus().toString());
    dest.setExitStatus(src.getExitStatus().toString());
}
}

```

## コントローラの実装

項目番	説明
(1)	@RestControllerを指定する。さらに@RequestMapping("job")により、web.xmlのサーブレットマッピングとあわせると、REST APIの基底パスはcontextName/api/v1/job/となる。
(2)	JobOperator、JobExplorerのフィールドインジェクションを記述する。
(3)	JobOperatorを使用して新規に非同期ジョブを起動する。 返り値としてジョブ実行IDを受け取り、REST クライアントに返却する。
(4)	JobExplorerを使用し、ジョブ実行IDを基にジョブの実行状態(JobExecution)を取得する。 あらかじめ設計された電文フォーマットに変換した上でRESTクライアントに返却する。

### 4.4.3.3.4. Web/バッチアプリケーションモジュール設定の統合

バッチアプリケーションモジュール(asyncbatch)は単体で動作可能なアプリケーションとして動作する。そのため、バッチアプリケーションモジュール(asyncbatch)は、Webアプリケーションモジュール(asyncapp-web)との間で競合・重複する設定が存在する。これらは、必要に応じて統合する必要がある。

#### 1. ログ設定ファイルlogback.xmlの統合

Web/バッチ間でLogback定義ファイルが複数定義されている場合、正常に動作しない。  
asyncbatch/src/main/resources/logback.xmlの記述内容はasyncapp-env/src/main/resources/の同ファイルに統合した上で削除する。

#### 2. データソース、MyBatis設定ファイルは統合しない

データソース、MyBatis設定ファイルの定義はWeb/バッチ間では、以下関係によりアプリケーションコンテキストの定義が独立するため、統合しない。

- バッチのasyncbatchモジュールはサーブレットに閉じたコンテキストとして定義される。
- Webのasyncapp-domain、asyncapp-envモジュールはアプリケーション全体で使用されるコンテキストとして定義される。

Webとバッチモジュールによるデータソース、MyBatis設定の相互参照

Webとバッチモジュールによるコンテキストのスコープが異なるため、特にWebモジュールからバッチのデータソース、MyBatis設定、Mapperインターフェースは参照できない。

RDBMSスキーマ初期化もそれぞれ異なるモジュールの設定に応じて独立して行われるため、相互干渉により意図しない初期化が行われないよう配慮すること。

#### REST コントローラ特有のCSRF対策設定

Webブランクプロジェクトの初期設定では、RESTコントローラに対しリクエストを送信するとCSRFエラーとしてジョブの実行が拒否される。そのため、ここでは以下方法によりCSRF対策を無効化した前提で説明している。

#### CSRF対策

ここで作成されるWebアプリケーションはインターネット上には公開されず、CSRFを攻撃手段として悪用しうる第三者からのRESTリクエスト送信が発生しない前提でCSRF対策を無効化している。実際のWebアプリケーションでは動作環境により要否が異なる点に注意すること。

#### 4.4.3.3.5. ビルド

Mavenコマンドでビルドし、warファイルを作成する。

```
$ cd asyncapp
$ ls
asyncbatch/  asyncapp-web/  pom.xml
$ mvn clean package
[INFO] Scanning for projects...
[INFO] -----
[INFO] Reactor Build Order:
[INFO]
[INFO] TERASOLUNA Server Framework for Java (5.x) Web Blank Multi Project (MyBatis3)
[INFO] TERASOLUNA Batch Framework for Java (5.x) Blank Project
[INFO] asyncapp-web
[INFO]
[INFO] -----
[INFO] Building TERASOLUNA Server Framework for Java (5.x) Web Blank Multi Project
(MyBatis3) 1.0-SNAPSHOT
[INFO] -----
(omitted)

[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] TERASOLUNA Server Framework for Java (5.x) Web Blank Multi Project (MyBatis3)
SUCCESS [ 0.226 s]
[INFO] TERASOLUNA Batch Framework for Java (5.x) Blank Project SUCCESS [ 6.481s]
[INFO] asyncapp-web ..... SUCCESS [ 5.400 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 12.597 s
[INFO] Finished at: 2017-02-10T22:32:43+09:00
[INFO] Final Memory: 38M/250M
[INFO] -----
$
```

#### 4.4.3.3.6. デプロイ

TomcatなどのWebコンテナを起動し、ビルドで生成されたwarファイルをデプロイする。 詳細な手順は割愛する。

#### 4.4.3.4. REST Clientによるジョブの起動と実行結果確認

ここではREST クライアントとしてcurlコマンドを使用し、非同期ジョブを起動する。

```
$ curl -v \
-H "Accept: application/json" -H "Content-type: application/json" \
-d '{"jobParams": "param1=value1"}' \
http://localhost:8080/asyncapp-web/api/v1/job/job01
* timeout on name lookup is not supported
* Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8088 (#0)
> POST /asyncapp-web/api/v1/job/job01 HTTP/1.1
> Host: localhost:8088
> User-Agent: curl/7.51.0
> Accept: application/json
> Content-type: application/json
> Content-Length: 30
>
* upload completely sent off: 30 out of 30 bytes
< HTTP/1.1 200
< X-Track: 0267db93977b4552880a4704cf3e4565
< Content-Type: application/json; charset=UTF-8
< Transfer-Encoding: chunked
< Date: Fri, 10 Feb 2017 13:55:46 GMT
<
{"jobName":"job01","jobParams":null,"jobExecutionId":3,"error":null,"errorMessage":null}* Curl_http_done: called premature == 0
* Connection #0 to host localhost left intact
$
```

上記より、ジョブ実行ID:**jobExecutionId = 3**として、ジョブが実行されていることが確認できる。  
続けてこのジョブ実行IDを使用し、ジョブの実行結果を取得する。

```
$ curl -v http://localhost:8080/asyncapp-web/api/v1/job/3
* timeout on name lookup is not supported
* Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8088 (#0)
> GET /asyncapp-web/api/v1/job/3 HTTP/1.1
> Host: localhost:8088
> User-Agent: curl/7.51.0
> Accept: */*
>
< HTTP/1.1 200
< X-Track: 7d94bf4d383745efb20cbf37cb6a8e13
< Content-Type: application/json; charset=UTF-8
< Transfer-Encoding: chunked
< Date: Fri, 10 Feb 2017 14:07:44 GMT
<
{"jobExecutionId":3,"jobName":"job01","stepExecutions":[{"stepExecutionId":5,"stepName":"job01.step01","status":"COMPLETED","failureExceptions":[],"exitStatus":{},"exitCode":COMPLETED,"exitDescription=","errorMessage":null}
]* Curl_http_done: called premature == 0
* Connection #0 to host localhost left intact
$
```

exitCode=COMPLETEDであることより、ジョブが正常終了していることが確認できる。



シェルスクリプトなどでcurlの実行結果を判定する場合

上記の例ではREST APIによる応答電文まで表示させている。curlコマンドでHTTPステータスのみを確認する場合はcurl -s URL -o /dev/null -w "%{http\_code}\n"とすることで、HTTPステータスが標準出力に表示される。

ただし、ジョブ実行IDはレスポンスボディ部のJSONを解析する必要があるため、必要に応じてREST クライアントアプリケーションを作成すること。

#### 4.4.4. How to extend

##### 4.4.4.1. ジョブの停止とリストート

非同期ジョブの停止・リストートは複数実行しているジョブの中から停止・リストートする必要がある。また、同名のジョブが並走している場合に、問題が発生しているジョブのみを対象にする必要もある。よって、対象とするジョブ実行が特定でき、その状態が確認できる必要がある。  
ここではこの前提を満たす場合、非同期実行の停止・リストートを行うための実装について説明する。

以降、[コントローラの実装](#)のJobControllerに対して、ジョブの停止(stop)やリストート(restart)を追加する方法について説明する。



ジョブの停止・リストートはJobOperatorを用いた実装をしなくても実施できる。  
詳細は[ジョブの管理](#)を参照し、目的に合う方式を検討すること。

##### 停止・リストートの実装例

```

// asyncapp/asyncapp-
web/src/main/java/org/terasoluna/batch/sample/app/api/JobController.java
package org.terasoluna.batch.sample.app.api;

// omitted

@RequestMapping("job")
@RestController
public class JobController {

    // omitted.

    @RequestMapping(value = "stop/{jobExecutionId}", method = RequestMethod.PUT)
    @Deprecated
    public ResponseEntity<JobOperationResource> stop(
        @PathVariable("jobExecutionId") Long jobExecutionId) {

        JobOperationResource responseResource = new JobOperationResource();
        responseResource.setJobExecutionId(jobExecutionId);
        boolean result = false;
        try {
            // (1)
            result = jobOperator.stop(jobExecutionId);
            if (!result) {
                responseResource.setErrorMessage("stop failed.");
                return ResponseEntity.badRequest().body(responseResource);
            }
            return ResponseEntity.ok().body(responseResource);
        } catch (NoSuchJobExecutionException | JobExecutionNotRunningException e) {
            responseResource.setError(e);
            return ResponseEntity.badRequest().body(responseResource);
        }
    }

    @RequestMapping(value = "restart/{jobExecutionId}",
                    method = RequestMethod.PUT)
    @Deprecated
    public ResponseEntity<JobOperationResource> restart(
        @PathVariable("jobExecutionId") Long jobExecutionId) {

        JobOperationResource responseResource = new JobOperationResource();
        responseResource.setJobExecutionId(jobExecutionId);
        try {
            // (2)
            Long id = jobOperator.restart(jobExecutionId);
            responseResource.setJobExecutionId(id);
            return ResponseEntity.ok().body(responseResource);
        } catch (JobInstanceAlreadyCompleteException |
                 NoSuchJobExecutionException | NoSuchJobException |
                 JobRestartException | JobParametersInvalidException e) {

```

```

        responseResource.setErrorMessage(e.getMessage());
        return ResponseEntity.badRequest().body(responseResource);
    }
}

// omitted.
}

```

#### コントローラによる停止・リスタート実装例

項目番	説明
(1)	JobOperator#stop()を呼び出すことにより、実行中のジョブに対し停止を指示する。
(2)	JobOperator#restart()を呼び出すことにより、異常終了・停止したステップから再実行させる。

#### 4.4.4.2. 複数起動

ここでの複数起動とは、Webコンテナを複数起動し、それぞれがジョブ要求を待ち受けることを指す。

非同期ジョブの実行管理は外部RDBMSによって行われるため、各アプリケーションの接続先となる外部RDBMSを共有することで、同一筐体あるいは別筐体にまたがって非同期ジョブ起動を待ち受けることができる。

用途としては特定のジョブに対する負荷分散や冗長化などがあげられる。しかし、[Webアプリケーションの実装](#)で述べたように、Webコンテナを複数起動し並列性を高めるだけでこれらの効果が容易に得られるわけではない。効果を得るためにには、一般的なWebアプリケーションと同様の対処が求められる場合がある。以下にその一例を示す。

- Webアプリケーションの特性上、1リクエスト処理はステートレスに動作するが、バッチの非同期実行はジョブの起動と結果の確認を合わせて設計しなければ、かえって障害耐性が低下する恐れもある。  
たとえば、ジョブ起動用Webコンテナを冗長化した場合でもクライアント側の障害によりジョブ起動後にジョブ実行IDをロストすることでジョブの途中経過や結果の確認は困難となる。
- 複数のWebコンテナにかかる負荷を分散させるために、クライアント側にリクエスト先を振り分ける機能を実装したり、ロードバランサを導入したりする必要がある。

このように、複数起動の適性は一概に定めることができない。そのため、目的と用途に応じてロードバランサの利用やWebクライアントによるリクエスト送信制御方式などを検討し、非同期実行アプリケーションの性能や耐障害性を落とさない設計が必要となる。

## 4.5. リスナー

### 4.5.1. Overview

リスナーとは、ジョブやステップを実行する前後に処理を挿入するためのインターフェースである。

本機能は、チャンクモデルとタスクレットモデルとで使い方が異なるため、それぞれについて説明する。

リスナーには多くのインターフェースがあるため、それぞれの役割について説明する。その後に、設定および実装方法について説明をする。

#### 4.5.1.1. リスナーの種類

Spring Batchでは、実際に多くのリスナーインターフェースが定義されている。ここではそのすべてを説明するのではなく、利用頻度が高いものを中心に扱う。

まず、リスナーは2種類に大別される。

##### *JobListener*

ジョブの実行に対して処理を挟み込むためのインターフェース

##### *StepListener*

ステップの実行に対して処理を挟み込むためのインターフェース

##### *JobListener*について

Spring Batchには、**JobListener**という名前のインターフェースは存在しない。

**StepListener**との対比のため、本ガイドラインでは便宜的に定義している。

Java Batch(jBatch)には、**javax.batch.api.listener.JobListener**というインターフェースが存在するので、実装時には間違えないように注意すること。また、**StepListener**もシグネチャが異なる同名インターフェース(**javax.batch.api.listener.StepListener**)が存在するので、同様に注意すること。



#### 4.5.1.1.1. JobListener

**JobListener**のインターフェースは、**JobExecutionListener**の1つのみとなる。

##### *JobExecutionListener*

ジョブの開始前、終了後に処理を挟み込む。

##### *JobExecutionListener*インターフェース

```
public interface JobExecutionListener {  
    void beforeJob(JobExecution jobExecution);  
    void afterJob(JobExecution jobExecution);  
}
```

#### 4.5.1.1.2. StepListener

**StepListener**のインターフェースは以下のように多くの種類がある。

## *StepListener*

以降に紹介する各種リスナーのマーカーインターフェース。

## *StepExecutionListener*

ステップ実行の開始前、終了後に処理を挟み込む。

### *StepExecutionListener*インターフェース

```
public interface StepExecutionListener extends StepListener {  
    void beforeStep(StepExecution stepExecution);  
    ExitStatus afterStep(StepExecution stepExecution);  
}
```

## *ChunkListener*

1つのチャunkを処理する前後と、エラーが発生した場合に処理を挟み込む。

### *ChunkListener*インターフェース

```
public interface ChunkListener extends StepListener {  
    static final String ROLLBACK_EXCEPTION_KEY = "sb_rollback_exception";  
    void beforeChunk(ChunkContext context);  
    void afterChunk(ChunkContext context);  
    void afterChunkError(ChunkContext context);  
}
```

#### *ROLLBACK\_EXCEPTION\_KEY*の用途

*afterChunkError*メソッドにて、発生した例外を取得したい場合に利用する。Spring Batchはチャunk処理中にエラーが発生した場合、*ChunkContext*に*sb\_rollback\_exception*というキー名で例外を格納した上で*ChunkListener*を呼び出すため、以下の要領でアクセスできる。

#### 使用例



```
public void afterChunkError(ChunkContext context) {  
    logger.error("Exception occurred while chunk. [context:{}]",  
    context,  
    context.getAttribute(ChunkListener.  
    ROLLBACK_EXCEPTION_KEY));  
}
```

## *ItemReadListener*

*ItemReader*が1件のデータを取得する前後と、エラーが発生した場合に処理を挟み込む。

## *ItemReadListener*インターフェース

```
public interface ItemReadListener<T> extends StepListener {  
    void beforeRead();  
    void afterRead(T item);  
    void onReadError(Exception ex);  
}
```

## *ItemProcessListener*

ItemProcessorが1件のデータを加工する前後と、エラーが発生した場合に処理を挟み込む。

## *ItemProcessListener*インターフェース

```
public interface ItemProcessListener<T, S> extends StepListener {  
    void beforeProcess(T item);  
    void afterProcess(T item, S result);  
    void onProcessError(T item, Exception e);  
}
```

## *ItemWriteListener*

ItemWriterが1つのチャンクを出力する前後と、エラーが発生した場合に処理を挟み込む。

## *ItemWriteListener*インターフェース

```
public interface ItemWriteListener<S> extends StepListener {  
    void beforeWrite(List<? extends S> items);  
    void afterWrite(List<? extends S> items);  
    void onWriteError(Exception exception, List<? extends S> items);  
}
```

本ガイドラインでは、以下のリスナーについては説明をしない。

- リトライ系リスナー
- スキップ系リスナー



これらのリスナーは例外ハンドリングでの使用を想定したものであるが、本ガイドラインではこれらのリスナーを用いた例外ハンドリングは行わない方針である。詳細は、[例外ハンドリング](#)を参照。

## 4.5.2. How to use

リスナーの実装と設定方法について説明する。

### 4.5.2.1. リスナーの実装

リスナーの実装と設定方法について説明する。

1. リスナーインターフェースを **implements** して実装する。

2. コンポーネントにメソッドベースでアノテーションを付与して実装する。

どちらで実装するかは、リスナーの役割に応じて選択する。基準は後述する。

#### 4.5.2.1.1. インターフェースを実装する場合

各種リスナーインターフェースを `implements` して実装する。必要に応じて、複数のインターフェースを同時に実装してもよい。以下に実装例を示す。

*JobExecutionListener* の実装例

```
@Component
public class JobExecutionLoggingListener implements JobExecutionListener { // (1)

    private static final Logger logger =
        LoggerFactory.getLogger(JobExecutionLoggingListener.class);

    @Override
    public void beforeJob(JobExecution jobExecution) { // (2)
        // do nothing.
    }

    @Override
    public void afterJob(JobExecution jobExecution) { // (3)

        logger.info("job finished.[JobName:{}][ExitStatus:{}]",
            jobExecution.getJobInstance().getJobName(),
            jobExecution.getExitStatus().getExitCode()); // (4)

        // per step execution
        // (5)
        jobExecution.getStepExecutions().forEach(stepExecution -> {
            Object errorItem = stepExecution.getExecutionContext().get("ERROR_ITEM");
            if (errorItem != null) {
                logger.error("detected error on this item processing. " +
                    "[step:{}][item:{}]", stepExecution.getStepName(),
                    errorItem);
            }
        });
    }
}
```

## リスナーの設定例

```
<batch:job id="chunkJobWithListener" job-repository="jobRepository">
    <batch:step id="chunkJobWithListener.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="reader" processor="processor"
                writer="writer" commit-interval="10"/>
            <batch:listeners>
                <batch:listener ref="loggingEachProcessInStepListener"/>
            </batch:listeners>
        </batch:tasklet>
    </batch:step>
    <batch:listeners>
        <batch:listener ref="jobExecutionLoggingListener"/> <!-- (6) -->
    </batch:listeners>
</batch:job>
```

## 説明

項番	説明
(1)	JobExecutionListenerを implements して実装する。
(2)	JobExecutionListenerが定義している beforeJob メソッドを実装する。 この例では、ジョブ開始前には何もしない。
(3)	JobExecutionListenerが定義している afterJob メソッドを実装する。 この例では、ジョブ終了時にジョブの最終状態と例外情報をログ出力する。
(4)	ジョブ名と終了コードをINFOログ出力する。必要な情報は、引数の JobExecution からそれぞれ取得する。
(5)	ステップごとに発生した例外をログ出力する。引数の JobExecution から紐づく StepExecution を取得し実現する。 ここでは、例外の起因となった入力データを ExecutionContext から ERROR_ITEM というキーで取得している。 ExecutionContext に設定する例は ジョブ単位の例外ハンドリング を参照。
(6)	Bean定義の <listeners> タグで、(1)で実装したリスナーを設定する。 設定方法の詳細は、 <a href="#">リスナーの設定</a> で説明する。

### リスナーのサポートクラス

複数のリスナーインターフェースを implements した場合、処理が不要な部分についても空実装をする必要がある。この作業を簡略化するため、あらかじめ空実装を施したサポートクラスが Spring Batch には用意されている。インターフェースではなく、サポートクラスを活用してもよいが、その場合 implements ではなく extends になるため注意すること。



### サポートクラス

- [org.springframework.batch.core.listener.ItemListenerSupport](#)
- [org.springframework.batch.core.listener.StepListenerSupport](#)

#### 4.5.2.1.2. アノテーションを付与する場合

各種リスナーインターフェースに対応したアノテーションを付与する。必要に応じて、複数のアノテーションを同時に実装してもよい。

リスナーインターフェースとの対応表

リスナーインターフェース	アノテーション
JobExecutionListener	@beforeJob @afterJob
StepExecutionListener	@BeforeStep @AfterStep
ChunkListener	@BeforeChunk @AfterChunk @afterChunkError
ItemReadListener	@BeforeRead @AfterRead @OnReadError
ItemProcessListener	@beforeProcess @afterProcess @onProcessError
ItemWriteListener	@BeforeWrite @AfterWrite @OnWriteError

これらアノテーションはコンポーネント化された実装のメソッドに付与することで目的のスコープで動作する。以下に実装例を示す。

## アノテーションを付与したItemProcessorの実装例

```
@Component
public class AnnotationAmountCheckProcessor implements
    ItemProcessor<SalesPlanDetail, SalesPlanDetail> {

    private static final Logger logger =
        LoggerFactory.getLogger(AnnotationAmountCheckProcessor.class);

    @Override
    public SalesPlanDetail process(SalesPlanDetail item) throws Exception {
        if (item.getAmount().signum() == -1) {
            throw new IllegalArgumentException("amount is negative.");
        }
        return item;
    }

    // (1)
    /*
     * @BeforeProcess
     public void beforeProcess(Object item) {
         logger.info("before process. [Item :{}]", item);
     }
    */

    // (2)
    @AfterProcess
    public void afterProcess(Object item, Object result) {
        logger.info("after process. [Result :{}]", result);
    }

    // (3)
    @OnProcessError
    public void onProcessError(Object item, Exception e) {
        logger.error("on process error.", e);
    }
}
```

## リスナーの設定例

```
<batch:job id="chunkJobWithListenerAnnotation" job-repository="jobRepository">
    <batch:step id="chunkJobWithListenerAnnotation.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="reader"
                processor="annotationAmountCheckProcessor"
                writer="writer" commit-interval="10"/> <! -- (4) -->
        </batch:tasklet>
    </batch:step>
</batch:job>
```

## 説明

項目番	説明
(1)	アノテーションで実装する場合は、処理が必要なタイミングのアノテーションのみを付与すればよい。 この例では、ItemProcessの処理前には何もする必要がないため、 <code>@beforeProcess</code> を付与した実装は不要となる。
(2)	ItemProcessの処理後に行う処理を実装する。 この例では処理結果をログを出力している。
(3)	ItemProcessでエラーが発生したときの処理を実装する。 この例では発生した例外をログを出力している。
(4)	アノテーションでリスナー実装がされているItemProcessを <code>&lt;chunk&gt;</code> タグに設定する。 リスナーアンターフェースとは異なり、 <code>&lt;listener&gt;</code> タグで設定しなくとも、自動的にリスナーが登録される。

アノテーションを付与するメソッドの制約



アノテーションを付与するメソッドはどのようなメソッドでもよいわけではない。対応するリスナーアンターフェースのメソッドと、シグネチャを一致させる必要がある。この点は、各アノテーションのjavadocに明記されている。

*JobExecutionListener*をアノテーションで実装したときの注意



JobExecutionListenerは、他のリスナーとスコープが異なるため、上記の設定では自動的にリスナー登録がされない。そのため、`<listener>`タグで明示的に設定する必要がある。詳細は、[リスナーの設定](#)を参照。

Tasklet実装へのアノテーションによるリスナー実装

Tasklet実装へのアノテーションによるリスナー実装した場合、以下の設定では一切リスナーが起動しないため注意する。

Taskletの場合



```
<batch:job id="taskletJobWithListenerAnnotation" job-
repository="jobRepository">
    <batch:step id="taskletJobWithListenerAnnotation.step01">
        <batch:tasklet transaction-manager="jobTransactionManager"
            ref=
            "annotationSalesPlanDetailRegisterTasklet"/>
    </batch:step>
</batch:job>
```

タスクレットモデルの場合は、[インターフェースとアノテーションの使い分け](#)に従ってリスナーアンターフェースを利用するのがよい。

### 4.5.2.2. リスナーの設定

リスナーは、Bean定義の`<listeners>`.`<listener>`タグによって設定する。XMLスキーマ定義では様々な箇所に記述できるが、インターフェースの種類によっては意図とおり動作しないものが存在するため、以

下の位置に設定すること。

リスナーを設定する位置

```
<!-- for chunk mode -->
<batch:job id="chunkJob" job-repository="jobRepository">
    <batch:step id="chunkJob.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="(1)"
                processor="(1)"
                writer="(1)" commit-interval="10"/>
            <batch:listeners>
                <batch:listener ref="(2)"/>
            </batch:listeners>
        </batch:tasklet>
    </batch:step>
    <batch:listeners>
        <batch:listener ref="(3)"/>
    </batch:listeners>
</batch:job>

<!-- for tasklet mode -->
<batch:job id="taskletJob" job-repository="jobRepository">
    <batch:step id="taskletJob.step01">
        <batch:tasklet transaction-manager="jobTransactionManager" ref="tasklet">
            <batch:listeners>
                <batch:listener ref="(2)"/>
            </batch:listeners>
        </batch:tasklet>
    </batch:step>
    <batch:listeners>
        <batch:listener ref="(3)"/>
    </batch:listeners>
</batch:job>
```

設定値の説明

項番	説明
(1)	StepListenerに属するアノテーションによる実装を含んだコンポーネントを設定する。 アノテーションの場合、必然的にこの場所に設定することになる。
(2)	StepListenerに属するリスナーアンターフェース実装を設定する。
(3)	JobListenerに属するリスナーを設定する。 インターフェースとアノテーション、どちらの実装でもここに設定する必要がある。

#### 4.5.2.2.1. 複数リスナーの設定

<batch:listeners>タグには複数のリスナーを設定することができる。

複数のリスナーを登録したときに、リスナーがどのような順番で起動されるかを以下に示す。

- ItemProcessListener実装

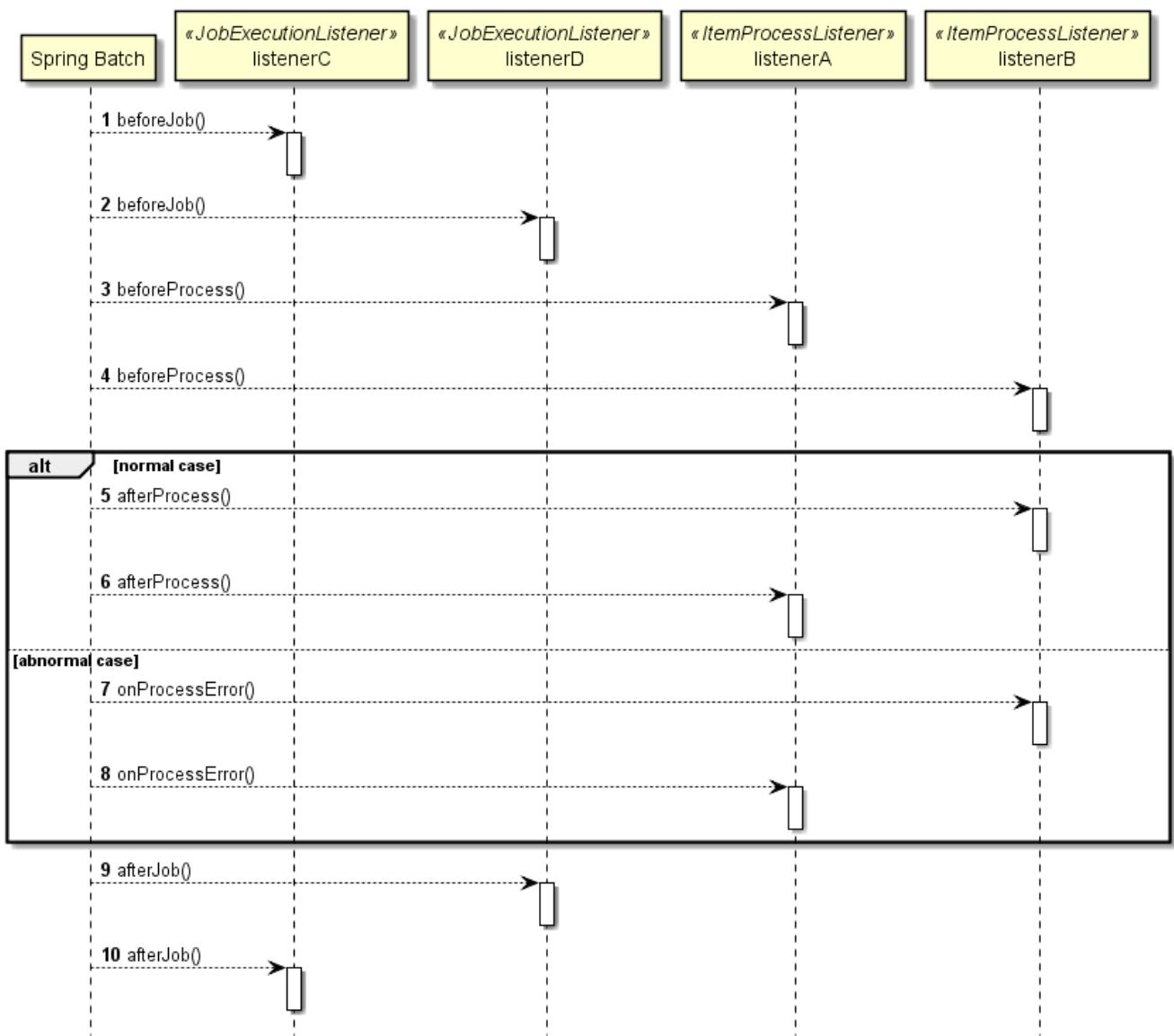
- listenerA, listenerB

- JobExecutionListener実装

- listenerC, listenerD

複数リスナーの設定例

```
<batch:job id="chunkJob" job-repository="jobRepository">
    <batch:step id="chunkJob.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="reader"
                processor="processor"
                writer="writer" commit-interval="10"/>
            <batch:listeners>
                <batch:listener ref="listenerA"/>
                <batch:listener ref="listenerB"/>
            </batch:listeners>
        </batch:tasklet>
    </batch:step>
    <batch:listeners>
        <batch:listener ref="listenerC"/>
        <batch:listener ref="listenerD"/>
    </batch:listeners>
</batch:job>
```



### リスナーの起動順序

- 前処理に該当する処理は、リスナーの登録順に起動される。
- 後処理またはエラー処理に該当する処理は、リスナー登録の逆順に起動される。

#### 4.5.2.3. インターフェースとアノテーションの使い分け

リスナーインターフェースとアノテーションによるリスナーの使い分けを説明する。

##### リスナーインターフェース

job、step、chunkにおいて共通する横断的な処理の場合に利用する。

##### アノテーション

ビジネスロジック固有の処理を行いたい場合に利用する。

原則として、ItemProcessorに対してのみ実装する。

# Chapter 5. データの入出力

## 5.1. トランザクション制御

### 5.1.1. Overview

本節では、ジョブにおけるトランザクション制御について以下の順序で説明する。

1. 一般的なバッチ処理におけるトランザクション制御のパターンについて
2. Spring Batchにおけるトランザクション制御
3. データベースやファイルといったリソースをトランザクショナルに処理するための方法

本機能は、チャンクモデルとタスクレットモデルとで使い方が異なるため、それぞれについて説明する。

#### 5.1.1.1. 一般的なバッチ処理におけるトランザクション制御のパターンについて

一般的に、バッチ処理は大量件数を処理するため、処理の終盤で何かしらのエラーが発生した場合に全件処理しなおしどなってしまうとバッチシステムのスケジュールに悪影響を与えててしまう。  
これを避けるために、1ジョブの処理内で一定件数ごとにトランザクションを確定しながら処理を進めていくことで、エラー発生時の影響を局所化することが多い。  
(以降、一定件数ごとにトランザクションを確定する方式を「中間コミット方式」、コミット単位にデータをひとまとめにしたものを作成する「チャンク」と呼ぶ。)

中間コミット方式のポイントを以下にまとめる。

1. エラー発生時の影響を局所化する。
  - 更新時にエラーが発生しても、エラー箇所直前のチャンクまで更新が確定している。
2. リソースを一定量しか使わない。
  - 処理対象データの大小問わず、チャンク分のリソースしか使用しないため安定する。

ただし、中間コミット方式があらゆる場面で有効な方法というわけではない。

システム内に一時的とはいっても処理済みデータと未処理データが混在することになる。その結果、リカバリ処理時に未処理データを識別する必要となるため、リカバリが複雑になる可能性がある。これを避けるには、中間コミット方式ではなく、全件を1トランザクションで確定させるしかない。  
(以降、全件を1トランザクションで確定する方式を「一括コミット方式」と呼ぶ。)

とはいっても、何千万件というような大量件数を一括コミット方式で処理してしまうと、コミットを行った際に全件をデータベース反映しようとして高負荷をかけてしまうような事態が発生する。そのため、一括コミット方式は小規模なバッチ処理には向いているが、大規模バッチで採用するには注意が必要となる。よって、この方法も万能な方法というわけではない。

つまり、「影響の局所化」と「リカバリの容易さ」はトレードオフの関係にある。「中間コミット方式」と「一括コミット方式」のどちらを使うかは、ジョブの性質に応じてどちらを優先すべきかを決定して欲しい。

もちろん、バッチシステム内のジョブすべてをどちらか一方で実現する必要はない。基本的には「中間コミット方式」を採用するが、特殊なジョブのみ「一括コミット方式」を採用する(または、その逆とする)

ことは自然である。

以下に、「中間コミット方式」と「一括コミット方式」のメリット・デメリット、採用ポイントをまとめます。

#### 方式別特徴一覧

コミット方式	メリット	デメリット	採用ポイント
中間コミット方式	エラー発生時の影響を局所化する	リカバリ処理が複雑になる可能性がある	大量データを一定のマシンリソースで処理したい場合
一括コミット方式	データの整合性を担保する	大量件数処理時に高負荷になる可能性がある	永続化リソースに対する処理結果をAll or Nothingとしたい場合 小規模のバッチ処理に向いている

データベースの同一テーブルへ入出力する際の注意点

データベースの仕組み上、コミット方式を問わず、同一テーブルへ入出力する処理で大量データを取り扱う際に注意が必要な点がある。

- 読み取り一貫性を担保するための情報が出力(UPDATEの発行)により失われた結果、入力(SELECT)にてエラーが発生することがある。

これを回避するには、以下の対策がある。



- 情報を確保する領域を大きくする。
  - 拡張する際には、リソース設計にて十分検討の上実施してほしい。
  - 拡張方法は使用するデータベースに依存するため、マニュアルを参照すること。
- 入力データを分割し多重処理を行う。
  - 多重処理については、[Partitioning Step \(多重処理\)](#)を参照。

## 5.1.2. Architecture

### 5.1.2.1. Spring Batchにおけるトランザクション制御

ジョブのトランザクション制御はSpring Batchがもつ仕組みを活用する。

以下に2種類のトランザクションを定義する。

#### フレームワークトランザクション

Spring Batchが制御するトランザクション

#### ユーザトランザクション

ユーザが制御するトランザクション

### 5.1.2.1.1. チャンクモデルにおけるトランザクション制御の仕組み

チャンクモデルにおけるトランザクション制御は、中間コミット方式のみとなる。一括コミット方式は実現できない。

チャンクモデルにおける一括コミット方式についてはJIRAにレポートされている。

<https://jira.spring.io/browse/BATCH-647>

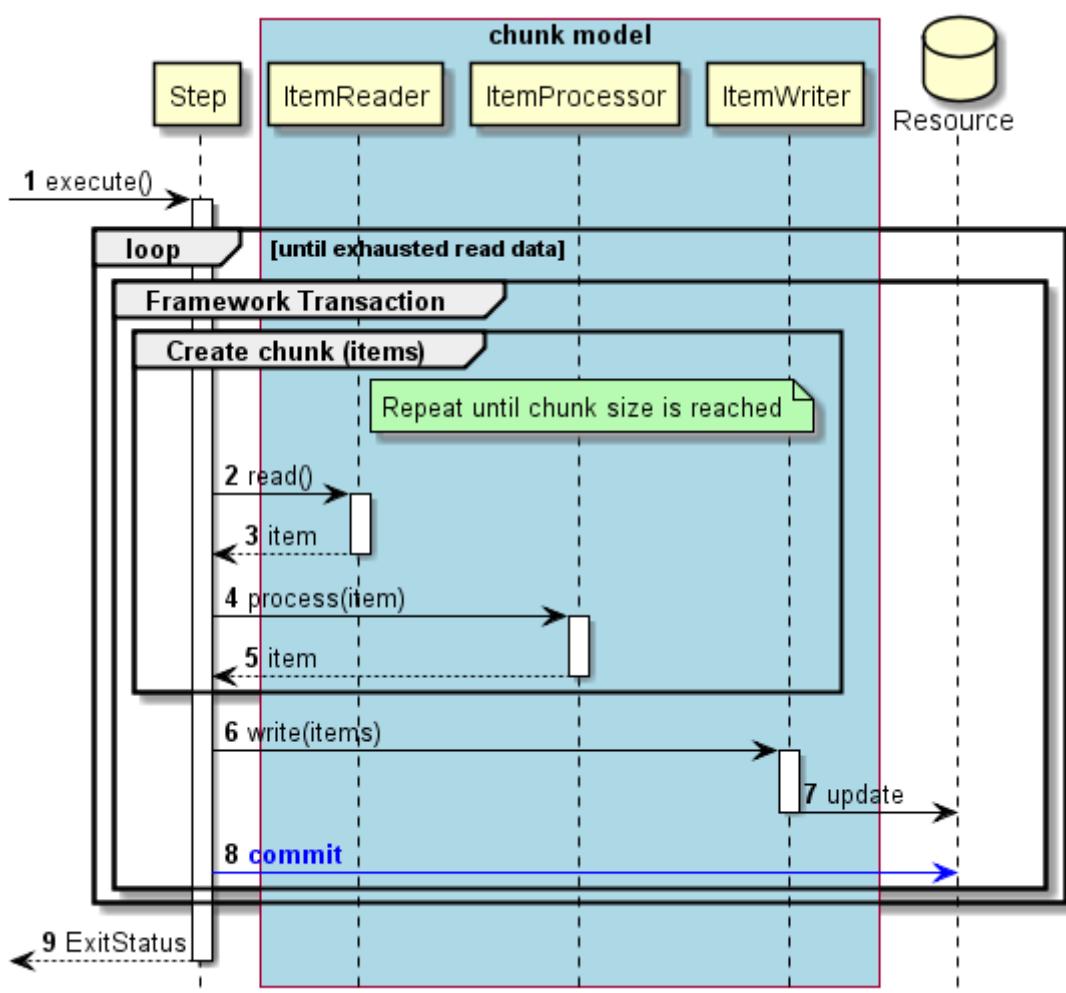


結果、`chunk completion policy`をカスタマイズしてチャンクサイズを動的に変更することで解決している。しかし、この方法では全データを1チャンクに格納してしまいメモリを圧迫してしまうため、方式として採用することはできない。

この方式の特徴は、チャンク単位にトランザクションが繰り返し行われることである。

正常系でのトランザクション制御

正常系でのトランザクション制御を説明する。



正常系のシーケンス図

シーケンス図の説明

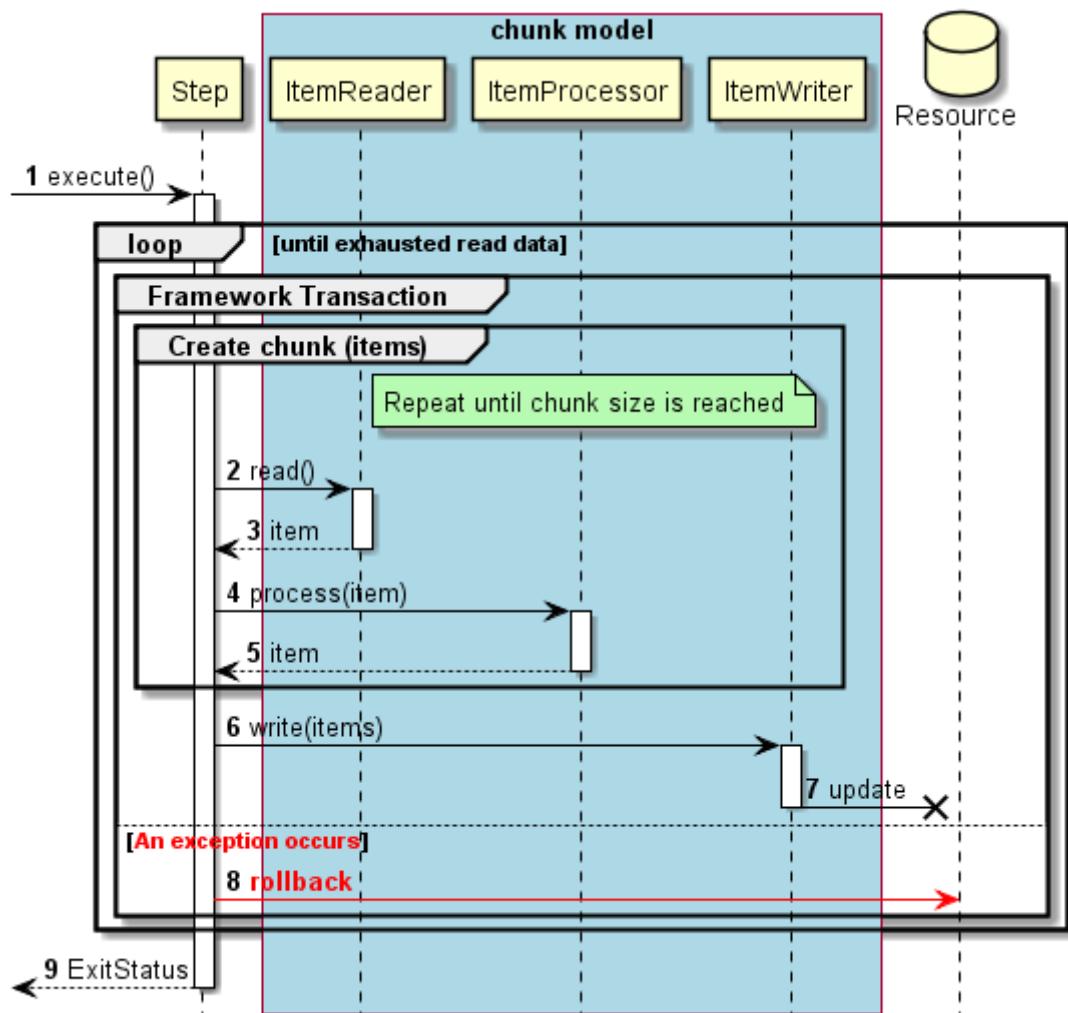
1. ジョブからステップが実行される。

- 入力データがなくなるまで、以降の処理を繰り返す。
- チャンク単位で、フレームワークトランザクションを開始する。
- チャンクサイズに達するまで2から5までの処理を繰り返す。

2. ステップは、ItemReaderから入力データを取得する。
3. ItemReaderは、ステップに入力データを返却する。
4. ステップは、ItemProcessorで入力データに対して処理を行う。
5. ItemProcessorは、ステップに処理結果を返却する。
6. ステップはチャンクサイズ分のデータをItemWriterで出力する。
7. ItemWriterは、対象となるリソースへ出力を行う。
8. ステップはフレームワークトランザクションをコミットする。

#### 異常系でのトランザクション制御

異常系でのトランザクション制御を説明する。



#### 異常系のシーケンス図

##### シーケンス図の説明

1. ジョブからステップが実行される。
  - 入力データがなくなるまで以降の処理を繰り返す。
  - チャンク単位でのフレームワークトランザクションを開始する。
  - チャンクサイズに達するまで2から5までの処理を繰り返す。
2. ステップは、ItemReaderから入力データを取得する。

3. **ItemReader**は、ステップに入力データを返却する。
4. ステップは、**ItemProcessor**で入力データに対して処理を行う。
5. **ItemProcessor**は、ステップに処理結果を返却する。
6. ステップはチャンクサイズ分のデータを**ItemWriter**で出力する。
7. **ItemWriter**は、対象となるリソースへ出力を行う。

2から7までの処理過程で例外が発生すると、

8. ステップはフレームワークトランザクションをロールバックする。

#### 5.1.2.1.2. タスクレットモデルにおけるトランザクション制御の仕組み

タスクレットモデルにおけるトランザクション制御は、一括コミット方式と中間コミット方式のいずれかを利用できる。

##### 一括コミット方式

Spring Batchがもつトランザクション制御の仕組みを利用する

##### 中間コミット方式

ユーザにてトランザクションを直接操作する

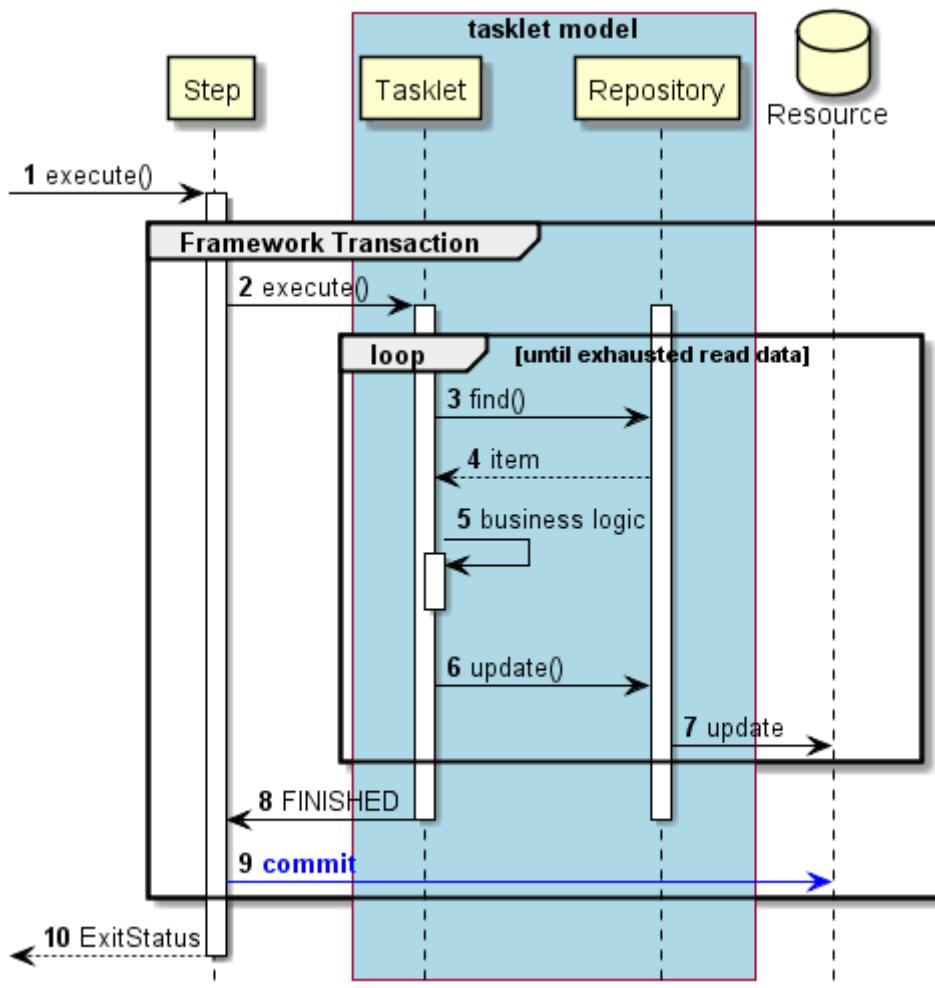
##### タスクレットモデルにおける一括コミット方式

Spring Batchがもつトランザクション制御の仕組みについて説明する。

この方式の特徴は、1つのトランザクション内で繰り返しデータ処理を行うことである。

##### 正常系でのトランザクション制御

正常系でのトランザクション制御を説明する。



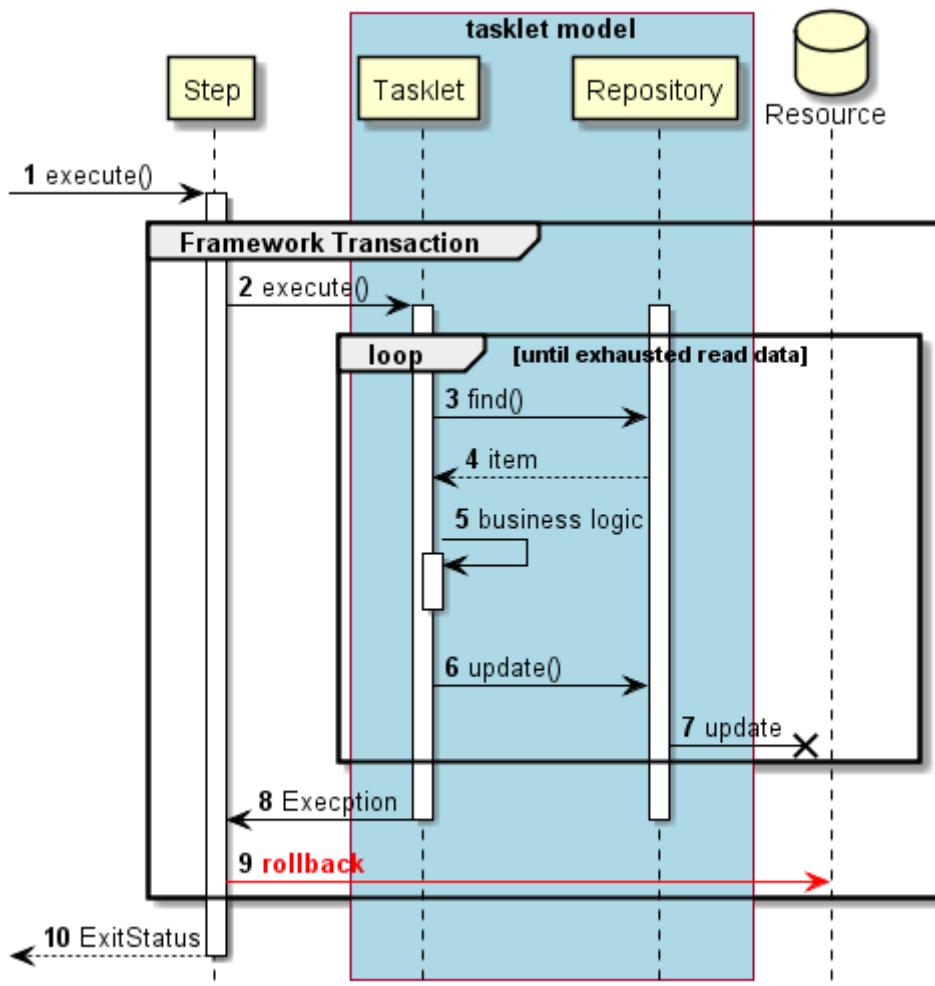
正常系のシーケンス図

#### シーケンス図の説明

1. ジョブからステップが実行される。
  - ・ステップはフレームワークトランザクションを開始する。
2. ステップはタスクレットを実行する。
  - ・入力データがなくなるまで3から7までの処理を繰り返す。
3. タスクレットは、Repositoryから入力データを取得する。
4. Repositoryは、タスクレットに入力データを返却する。
5. タスクレットは、入力データを処理する。
6. タスクレットは、Repositoryへ出力データを渡す。
7. Repositoryは、対象となるリソースへ出力を行う。
8. タスクレットはステップへ処理終了を返却する。
9. ステップはフレームワークトランザクションをコミットする。

#### 異常系でのトランザクション制御

異常系でのトランザクション制御を説明する。



異常系のシーケンス図

#### シーケンス図の説明

1. ジョブからステップが実行される。
  - ・ステップはフレームワークトランザクションを開始する。
2. ステップはタスクレットを実行する。
  - ・入力データがなくなるまで3から7までの処理を繰り返す。
3. タスクレットは、**Repository**から入力データを取得する。
4. **Repository**は、タスクレットに入力データを返却する。
5. タスクレットは、入力データを処理する。
6. タスクレットは、**Repository**へ出力データを渡す。
7. **Repository**は、対象となるリソースへ出力を行う。

2から7までの処理過程で例外が発生すると、

8. タスクレットはステップへ例外をスローする。
9. ステップはフレームワークトランザクションをロールバックする。

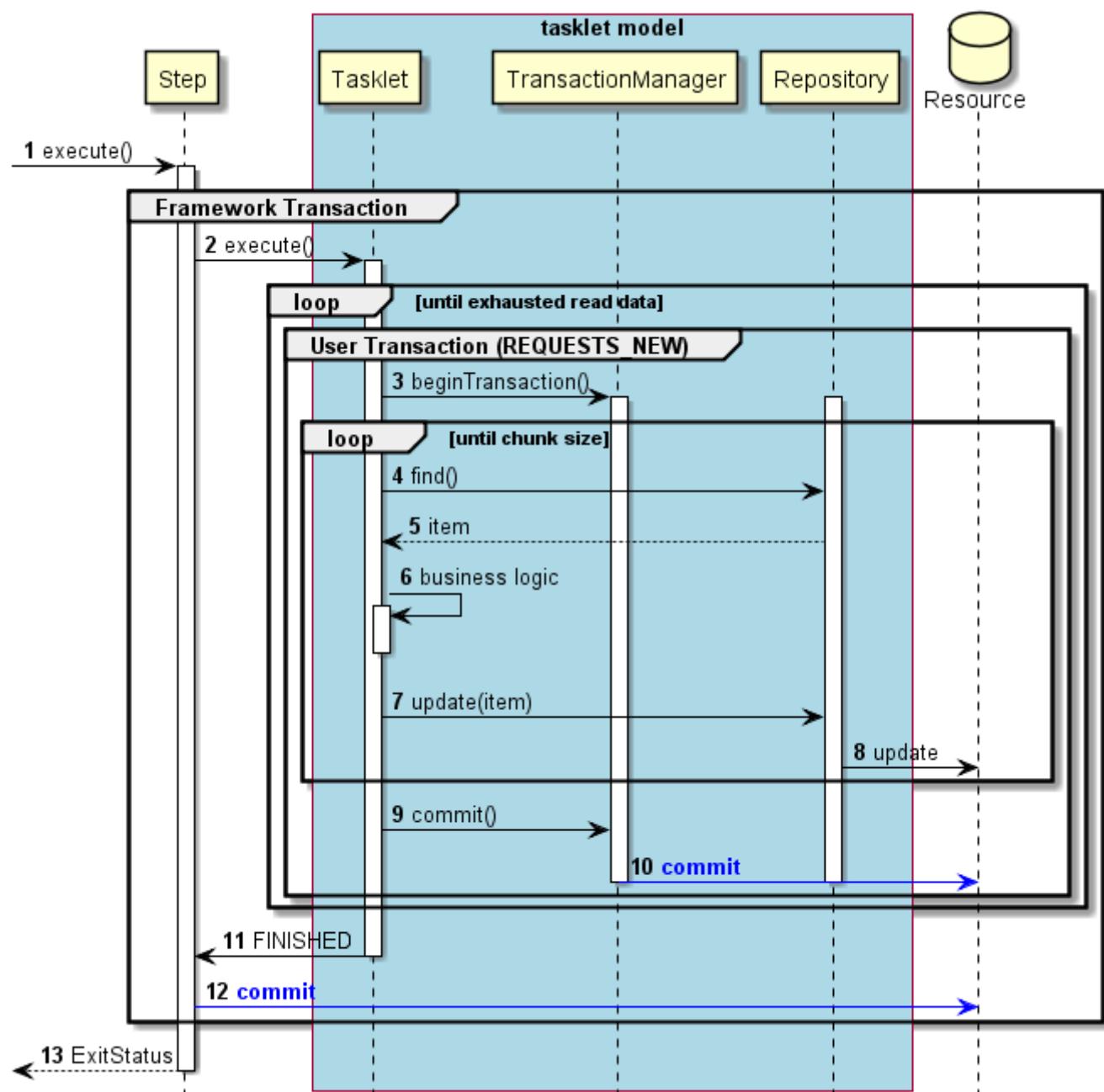
## タスクレットモデルにおける中間コミット方式

ユーザにてトランザクションを直接操作する仕組みについて説明する。

この方式の特徴は、フレームワークトランザクション内で新規のユーザトランザクションを開始して操作することである。

### 正常系でのトランザクション制御

正常系でのトランザクション制御を説明する。



### 正常系のシーケンス図

#### シーケンス図の説明

1. ジョブからステップが実行される。
  - ステップは フレームワークトランザクション を開始する。
2. ステップはタスクレットを実行する。

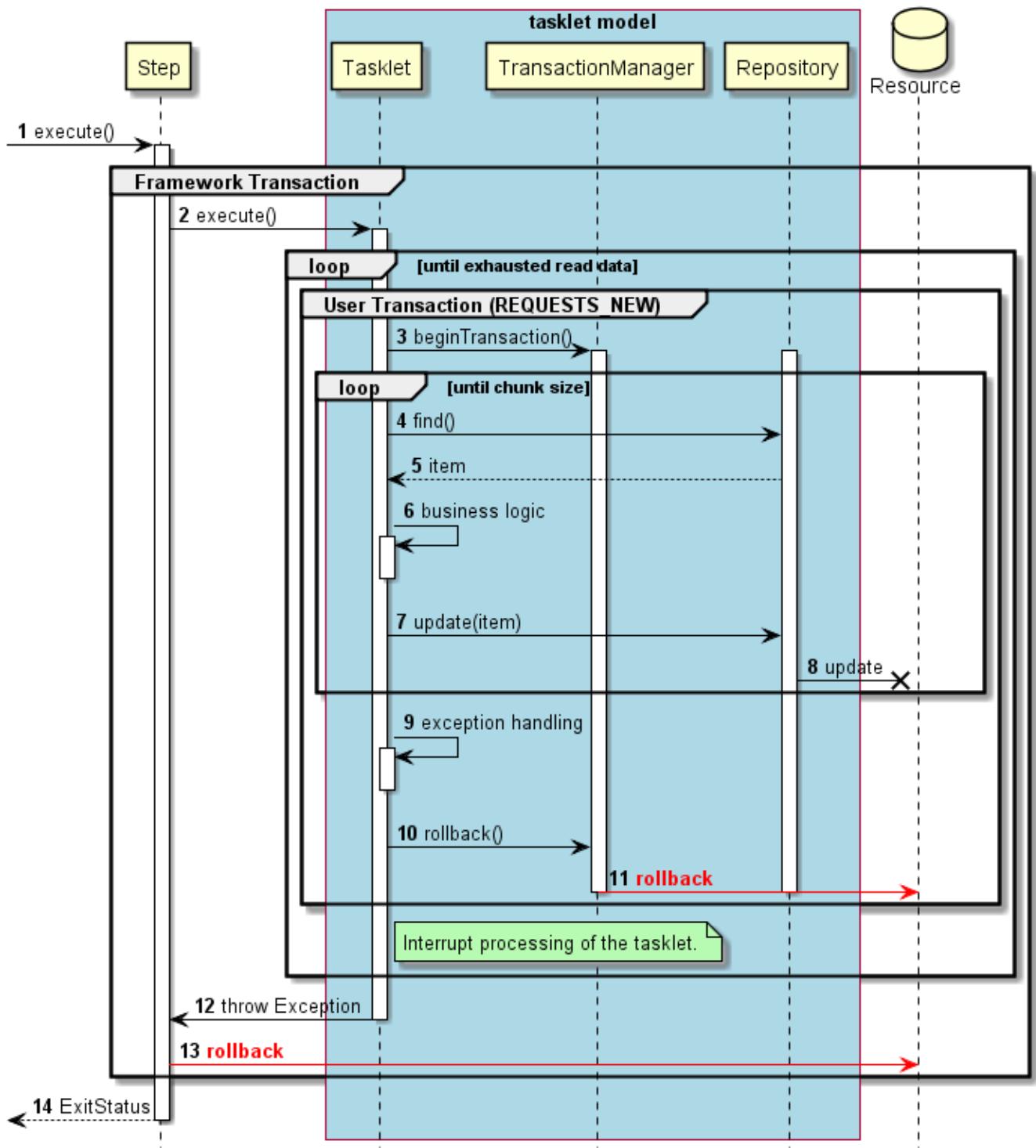
- 入力データがなくなるまで3から10までの処理を繰り返す。
3. タスクレットは、[TransacitonManager](#)よりユーザトランザクションを開始する。
    - フレームワークのトランザクションと分離させるため、`REQUIRES_NEW`でユーザトランザクションを実行する。
    - チャンクサイズに達するまで4から6までの処理を繰り返す。
  4. タスクレットは、[Repository](#)から入力データを取得する。
  5. [Repository](#)は、タスクレットに入力データを返却する。
  6. タスクレットは、入力データを処理する。
  7. タスクレットは、[Repository](#)へ出力データを渡す。
  8. [Repository](#)は、対象となるリソースへ出力を行う。
  9. タスクレットは、[TransacitonManager](#)によりユーザトランザクションのコミットを実行する。
  10. [TransacitonManager](#)は、対象となるリソースへコミットを発行する。
  11. タスクレットはステップへ処理終了を返却する。
  12. ステップはフレームワークトランザクションをコミットする。



ここでは1件ごとにリソースへ出力しているが、チャンクモデルと同様に、チャンク単位で一括更新し処理スループットの向上を狙うことも可能である。その際に、[SqlSessionTemplate](#)の[executorType](#)を[BATCH](#)に設定することで、BatchUpdateを利用することもできる。これは、MyBatisのItemWriterを利用する場合と同様の動作になるため、MyBatisのItemWriterを利用して更新してもよい。MyBatisのItemWriterについて、詳細は[ItemWriterにおけるデータベースアクセス](#)を参照。

#### 異常系でのトランザクション制御

異常系でのトランザクション制御を説明する。



異常系のシーケンス図

#### シーケンス図の説明

- ジョブからステップが実行される。
  - ステップはフレームワークトランザクションを開始する。
- ステップはタスクレットを実行する。
  - 入力データがなくなるまで3から11までの処理を繰り返す。
- タスクレットは、**TransacitonManager**よりユーザトランザクションを開始する。
  - フレームワークのトランザクションと分離させるため、**REQUIRES\_NEW**でユーザトランザクシ

ョンを実行する。

- ・チャンクサイズに達するまで4から6までの処理を繰り返す。
- タスクレットは、**Repository**から入力データを取得する。
  - Repository**は、タスクレットに入力データを返却する。
  - タスクレットは、入力データを処理する。
  - タスクレットは、**Repository**へ出力データを渡す。
  - Repository**は、対象となるリソースへ出力を行う。

3から8までの処理過程で例外が発生すると、

- タスクレットは、発生した例外に対する処理を行う。
- タスクレットは、**TransacitonManager**によりユーザトランザクションのロールバックを実行する。
- TransacitonManager**は、対象となるリソースへロールバックを発行する。
- タスクレットはステップへ例外をスローする。
- ステップはフレームワークトランザクションをロールバックする。

#### 処理の継続について



ここでは、例外をハンドリングして処理をロールバック後、処理を異常終了しているが、継続して次のチャンクを処理することも可能である。いずれの場合も、途中でエラーが発生したことをステップのステータス・終了コードを変更することで後続の処理に通知する必要がある。

#### フレームワークトランザクションについて



ここでは、ユーザトランザクションをロールバック後に例外をスローしてジョブを異常終了させているが、ステップへ処理終了を返却しジョブを正常終了させることも出来る。この場合、フレームワークトランザクションは、コミットされる。

##### 5.1.2.1.3. モデル別トランザクション制御の選定方針

TERASOLUNA Batch 5.xの基盤となるSpring Batchでは、チャンクモデルでは中間コミット方式しか実現できない。しかし、タスクレットモデルでは、中間コミット方式、一括コミット方式のいずれも実現できる。

よって、TERASOLUNA Batch 5.xでは、一括コミット方式が必要な際は、タスクレットモデルにて実装する。

##### 5.1.2.2. 起動方式ごとのトランザクション制御の差

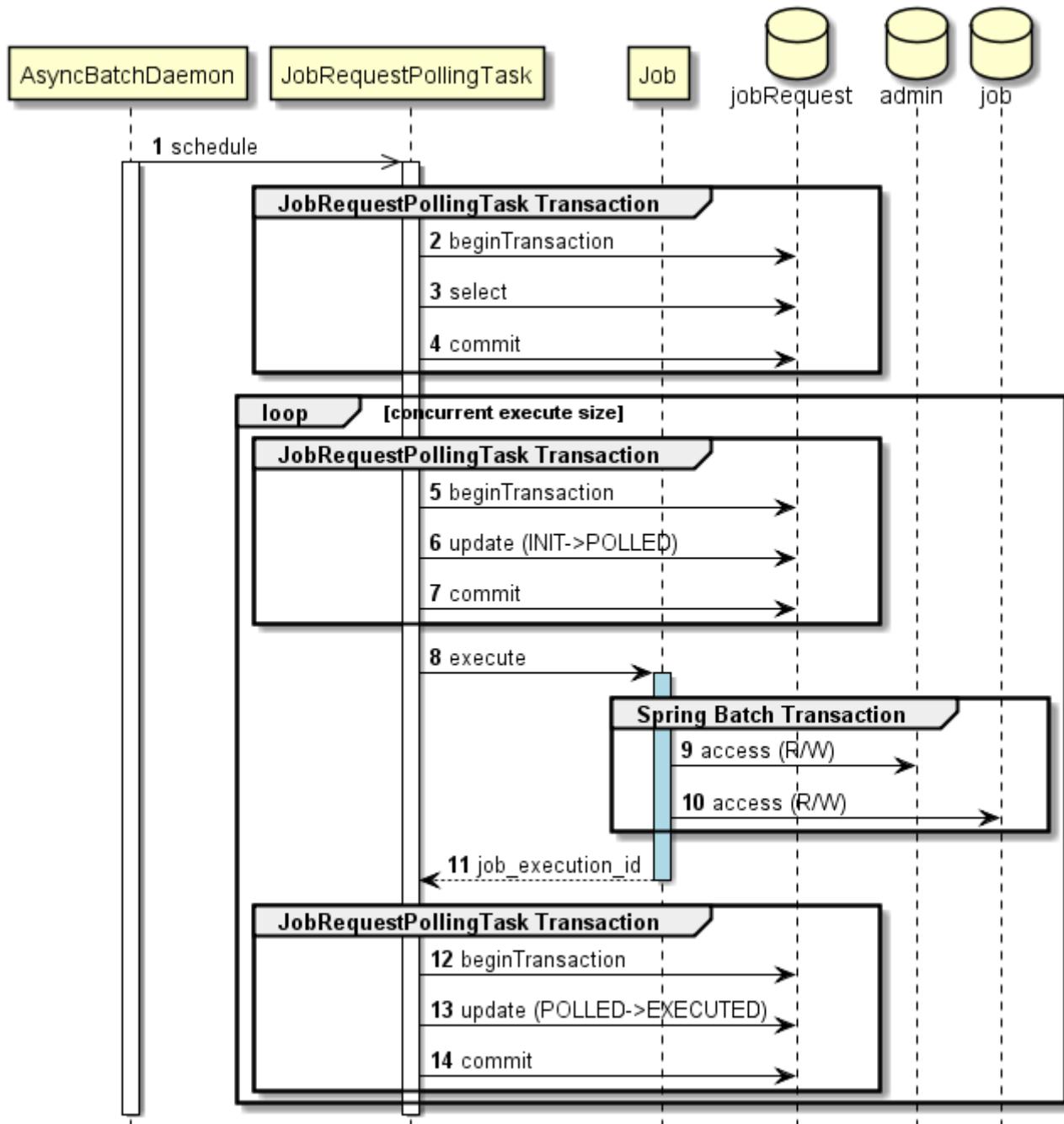
起動方式によってはジョブの起動前後にSpring Batchの管理外となるトランザクションが発生する。ここでは、2つの非同期実行処理方式におけるトランザクションについて説明する。

###### 5.1.2.2.1. DBポーリングのトランザクションについて

DBポーリングが行うジョブ要求テーブルへの処理については、Spring Batch管理外のトランザクション処理が行われる。また、ジョブで発生した例外については、ジョブ内で対応が完結するた

め、`JobRequestPollTask`が行うトランザクションには影響を与えない。

下図にトランザクションに焦点を当てた簡易的なシーケンス図を示す。



### DBポーリング処理のトランザクション

#### シーケンス図の説明

1. 非同期バッチデーモンで`JobRequestPollTask`が周期実行される。
2. `JobRequestPollTask`は、Spring Batch管理外のトランザクションを開始する。
3. `JobRequestPollTask`は、ジョブ要求テーブルから非同期実行対象ジョブを取得する。
4. `JobRequestPollTask`は、Spring Batch管理外のトランザクションをコミットする。
5. `JobRequestPollTask`は、Spring Batch管理外のトランザクションを開始する。
6. `JobRequestPollTask`は、ジョブ要求テーブルのポーリングステータスをINITからPOLLEDへ更新する。

7. `JobRequestPollTask`は、Spring Batch管理外のトランザクションをコミットする。
8. `JobRequestPollTask`は、ジョブを実行する。
9. ジョブ内では、管理用DB(`JobRepository`)へのトランザクション管理はSpring Batchが行う。
10. ジョブ内では、ジョブ用DBへのトランザクション管理はSpring Batchが行う。
11. `JobRequestPollTask`に`job_execution_id`が返却される
12. `JobRequestPollTask`は、Spring Batch管理外のトランザクションを開始する。
13. `JobRequestPollTask`は、ジョブ要求テーブルのポーリングステータスをPOLLEDからEXECUTEへ更新する。
14. `JobRequestPollTask`は、Spring Batch管理外のトランザクションをコミットする。

*SELECT*発行時のコミットについて

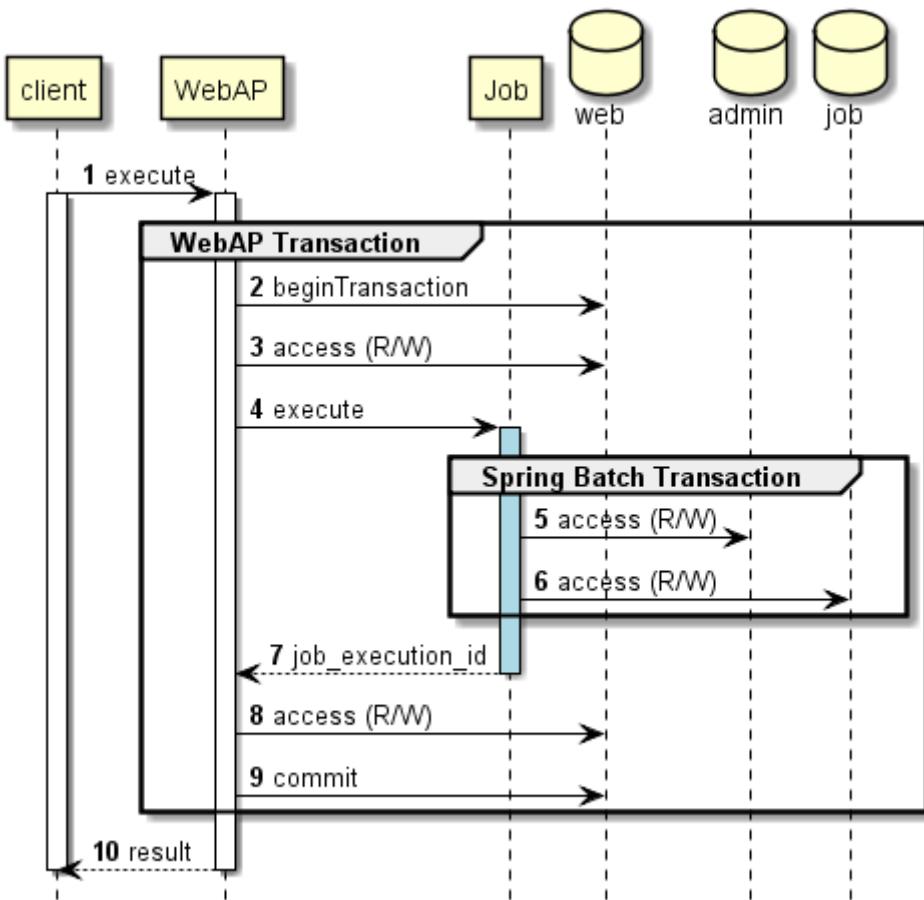


データベースによっては、*SELECT*発行時に暗黙的にトランザクションを開始する場合がある。そのため、明示的にコミットを発行することでトランザクションを確定させ、他のトランザクションと明確に区別し影響を与えないようにしている。

#### 5.1.2.2.2. WebAPサーバ処理のトランザクションについて

WebAPが対象とするリソースへの処理については、Spring Batch管理外のトランザクション処理が行われる。また、ジョブで発生した例外については、ジョブ内で対応が完結するため、WebAPが行うトランザクションには影響を与えない。

下図にトランザクションに焦点を当てた簡易的なシーケンス図を示す。



WebAPサーバ処理のトランザクション

#### シーケンス図の説明

1. クライアントからリクエストによりWebAPの処理が実行される
2. WebAPは、Spring Batch管理外のトランザクションを開始する。
3. WebAPは、ジョブ実行前にWebAPでのリソースに対して読み書きを行う。
4. WebAPは、ジョブを実行する。
5. ジョブ内では、管理用DB(**JobRepository**)へのトランザクション管理はSpring Batchが行う。
6. ジョブ内では、ジョブ用DBへのトランザクション管理はSpring Batchが行う。
7. WebAPに`job_execution_id`が返却される
8. WebAPは、ジョブ実行後にWebAPでのリソースに対して読み書きを行う。
9. WebAPは、Spring Batch管理外のトランザクションをコミットする。
10. WebAPは、クライアントにレスポンスを返す。

#### 5.1.3. How to use

ここでは、1ジョブにおけるトランザクション制御について、以下の場合に分けて説明する。

- **単一データソースの場合**
- **複数データソースの場合**

データソースとは、データの格納先(データベース、ファイル等)を指す。単一データソースとは1つの

データソースを、複数データソースとは2つ以上のデータソースを指す。

单一データソースを処理するケースは、データベースのデータを加工するケースが代表的である。  
複数データソースを処理するケースは、以下のようにいくつかバリエーションがある。

- ・複数のデータベースの場合
- ・データベースとファイルの場合

#### 5.1.3.1. 単一データソースの場合

1つのデータソースに対して入出力するジョブのトランザクション制御について説明する。

以下にTERASOLUNA Batch 5.xでの設定例を示す。

データソースの設定(*META-INF/spring/launch-context.xml*)

```
<!-- Job-common definitions -->
<bean id="jobDataSource" class="org.apache.commons.dbcp2.BasicDataSource"
    destroy-method="close"
    p:driverClassName="${jdbc.driver}"
    p:url="${jdbc.url}"
    p:username="${jdbc.username}"
    p:password="${jdbc.password}"
    p:maxTotal="10"
    p:minIdle="1"
    p:maxWaitMillis="5000"
    p:defaultAutoCommit="false" />
```

トランザクションマネージャの設定(*META-INF/spring/launch-context.xml*)

```
<!-- (1) -->
<bean id="jobTransactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
    p:dataSource-ref="jobDataSource"
    p:rollbackOnCommitFailure="true" />
```

項目番号	説明
(1)	トランザクションマネージャのBean定義 データソースは上記で定義した <b>jobDataSource</b> を設定する。 コミットに失敗した場合はロールバックをするように設定済み。

##### 5.1.3.1.1. トランザクション制御の実施

ジョブモデルおよびコミット方式により制御方法が異なる。

チャンクモデルの場合

チャンクモデルの場合は、中間コミット方式となり、Spring Batchにトランザクション制御を委ねる。  
ユーザにて制御することは一切行わないようとする。

## 設定例(ジョブ定義)

```
<batch:job id="jobSalesPlan01" job-repository="jobRepository">
    <batch:step id="jobSalesPlan01.step01">
        <batch:tasklet transaction-manager="jobTransactionManager"> <!-- (1) -->
            <batch:chunk reader="detailCSVReader"
                writer="detailWriter"
                commit-interval="10" /> <!-- (2) -->
        </batch:tasklet>
    </batch:step>
</batch:job>
```

項番	説明
(1)	<batch:tasklet>タグのtransaction-manager属性に定義済みの jobTransactionManagerを設定する。 ここに設定したトランザクションマネージャでチャンクコミット方式のトランザクションを制御する。
(2)	commit-interval属性にチャンクサイズを設定する。この例では10件処理するごとに 1回コミットが発行される。

## タスクレットモデルの場合

タスクレットモデルの場合は、一括コミット方式、中間コミット方式でトランザクション制御の方法が異なる。

### 一括コミット方式

Spring Batchにトランザクション制御を委ねる。

## 設定例(ジョブ定義)

```
<batch:job id="jobSalesPlan01" job-repository="jobRepository">
    <batch:step id="jobSalesPlan01.step01">
        <!-- (1) -->
        <batch:tasklet transaction-manager="jobTransactionManager"
            ref="salesPlanSingleTranTask" />
    </batch:step>
</batch:job>
```

項番	説明
(1)	<batch:tasklet>タグのtransaction-manager属性に定義済みの jobTransactionManagerを設定する。 ここに設定したトランザクションマネージャで一括コミット方式のトランザクションを制御する。

### 中間コミット方式

ユーザにてトランザクション制御を行う。

- 処理の途中でコミットを発行する場合は、TransacitonManagerをInjectして手動で行う。

## 設定例(ジョブ定義)

```
<batch:job id="jobSalesPlan01" job-repository="jobRepository">
    <batch:step id="jobSalesPlan01.step01">
        <!-- (1) -->
        <batch:tasklet transaction-manager="jobTransactionManager"
            ref="salesPlanChunkTranTask" />
    </batch:step>
</batch:job>
```

## 実装例

```
@Component()
public class SalesPlanChunkTranTask implements Tasklet {

    @Inject
    ItemStreamReader<SalesPlanDetail> itemReader;

    // (2)
    @Inject
    @Named("jobTransactionManager")
    PlatformTransactionManager transactionManager;

    @Inject
    SalesPlanDetailRepository repository;

    private static final int CHUNK_SIZE = 10;

    @Override
    public RepeatStatus execute(StepContribution contribution,
                               ChunkContext chunkContext) throws Exception {

        DefaultTransactionDefinition definition = new DefaultTransactionDefinition();
        definition.setPropagationBehavior(TransactionDefinition
            .PROPAGATION_REQUIRES_NEW); // (3)
        TransactionStatus status = null;

        try {
            // omitted

            itemReader.open(executionContext);

            while ((item = itemReader.read()) != null) {

                if (count % CHUNK_SIZE == 0) {
                    status = transactionManager.getTransaction(definition); // (4)
                }
                count++;

                // omitted
            }
        } catch (Exception e) {
            if (status != null) {
                status.rollback();
            }
            throw e;
        }
    }
}
```

```

        repository.create(item);
        if (count % CHUNK_SIZE == 0) {
            transactionManager.commit(status); // (5)
        }
    }
} catch (Exception e) {
    logger.error("Exception occurred while reading.", e);
    transactionManager.rollback(status); // (6)
    throw e;
} finally {
    if (!status.isCompleted()) {
        transactionManager.commit(status); // (7)
    }
    itemReader.close();
}

return RepeatStatus.FINISHED;
}
}

```

項番	説明
(1)	<batch:tasklet>タグのtransaction-manager属性に定義済みの jobTransactionManagerを設定する。 1つのトランザクションマネージャをフレームワーク/ユーザの両方で利用するが、以降の要領により独立して扱うことが出来る。
(2)	トランザクションマネージャをInjectする。 @NamedアノテーションでjobTransactionManagerを指定して利用するBeanを特定させる。
(3)	フレームワークのトランザクションとは分離させるため、PROPAGATION_REQUIRES_NEWを指定する。
(4)	チャックの開始時にトランザクションを開始する。
(5)	チャック終了時にトランザクションをコミットする。
(6)	例外発生時にはトランザクションをロールバックする。
(7)	最後のチャックについて、トランザクションをコミットする。

*TransacitonManager*のPropagationについて



タスクレットモデルでは、Spring Batchが制御しているトランザクション内で新たにトランザクション制御を行う。そのため、InjectするTransacitonManagerのPropagationを REQUIRES\_NEW にする必要がある。

*ItemWriter*による更新



上記の例では、Repositoryを使用しているが、ItemWriterを利用してデータを更新することもできる。ItemWriterを利用することで実装がシンプルになる効果があり、特にファイルを更新する場合はFlatFileItemWriterを利用するとよい。

### 5.1.3.1.2. 非トランザクショナルなデータソースに対する補足

ファイルの場合はトランザクションの設定や操作は不要である。

`FlatFileItemWriter`を利用する場合、擬似的なトランザクション制御が行える。これは、リソースへの書き込みを遅延し、コミットタイミングで実際に書き出すことで実現している。正常時にはチャunkサイズに達したときに、実際のファイルにチャunk分データを出力し、例外が発生するとそのチャunkのデータ出力が行われない。

`FlatFileItemWriter`は、`transactional`プロパティでトランザクション制御の有無を切替えられる。デフォルトはtrueでトランザクション制御が有効になっている。`transactional`プロパティがfalseの場合、`FlatFileItemWriter`は、トランザクションとは無関係にデータの出力を行う。

一括コミット方式を採用する場合、`transactional`プロパティをfalseにすることを推奨する。上記の説明にあるとおりコミットのタイミングでリソースへ書き出すため、それまではメモリ内に全出力分のデータを保持することになる。そのため、データ量が多い場合にはメモリ不足になりエラーとなる可能性が高くなるためである。

ファイルしか扱わないジョブにおける`TransacitonManager`の設定について  
以下に示すジョブ定義のように、`batch:tasklet`の`transaction-manager`属性はxsdスキーマにおいて必須のため省略できない。

#### *TransacitonManager*設定箇所の抜粋

```
<batch:tasklet transaction-manager="jobTransactionManager">  
<batch:chunk reader="reader" writer="writer" commit-interval="100" />  
</batch:tasklet>
```

そのため、`jobTransactionManager`を常に指定すること。この時、以下の挙動となる。

- `transactional`がtrueの場合
  - 指定した`TransacitonManager`に同期してリソースに出力する。
- `transactional`がfalseの場合
  - 指定した`TransacitonManager`のトランザクション処理は空振りし、トランザクションと関係なくリソースに出力する。



この時、`jobTransactionManager`が参照するリソース(たとえば、データベース)に対してトランザクションが発行されるが、テーブルアクセスは伴わないので実害がない。

また、実害がある場合や空振りでも参照するトランザクションを発行したくない場合は、リソースを必要としない`ResourcelessTransactionManager`を使用することができる。

#### *ResourcelessTransactionManager*の使用例

```
<batch:tasklet transaction-manager="resourcelessTransactionManager">  
<batch:chunk reader="reader" writer="writer" commit-interval="100" />  
</batch:tasklet>  
  
<bean id="resourcelessTransactionManager"  
class="org.springframework.batch.support.transaction.ResourcelessTrans  
actionManager"/>
```

### 5.1.3.2. 複数データソースの場合

複数データソースに対して入出力するジョブのトランザクション制御について説明する。入力と出力で考慮点が異なるため、これらを分けて説明する。

#### 5.1.3.2.1. 複数データソースからの取得

複数データソースからのデータを取得する場合、処理の軸となるデータと、それに付随する追加データを分けて取得する。以降は、処理の軸となるデータを処理対象レコード、それに付随する追加データを付随データと呼ぶ。

Spring Batchの構造上、ItemReaderは1つのリソースから処理対象レコードを取得することを前提としているためである。これは、リソースの種類を問わず同じ考え方となる。

## 1. 処理対象レコードの取得

- ItemReaderにて取得する。

## 2. 付随データの取得

- 付随データは、そのデータに対する変更の有無と件数に応じて、以下の取得方法を選択する必要がある。これは、逐一ではなく、併用してもよい。
  - ステップ実行前に一括取得
  - 処理対象レコードに応じて都度取得

ステップ実行前に一括取得する場合

以下を行うListenerを実装し、以降のStepからデータを参照する。

- データを一括して取得する
- スコープがJobまたはStepのBeanに情報を格納する
  - Spring BatchのExecutionContextを活用してもよいが、可読性や保守性のために別途データ格納用のクラスを作成してもよい。ここでは、簡単のためExecutionContextを活用した例で説明する。

マスターデータなど、処理対象データに依存しないデータを読み込む場合にこの方法を採用する。ただし、マスターデータと言えど、メモリを圧迫するような大量件数が対象である場合は、都度取得したほうがよいかを検討すること。

一括取得するListenerの実装

```
@Component
// (1)
public class BranchMasterReadStepListener extends StepExecutionListenerSupport {

    @Inject
    BranchRepository branchRepository;

    @Override
    public void beforeStep(StepExecution stepExecution) { // (2)

        List<Branch> branches = branchRepository.findAll(); // (3)

        Map<String, Branch> map = branches.stream()
            .collect(Collectors.toMap(Branch::getBranchId,
                UnaryOperator.identity())); // (4)

        stepExecution.getExecutionContext().put("branches", map); // (5)
    }
}
```

## 一括取得するListenerの定義

```
<batch:job id="outputAllCustomerList01" job-repository="jobRepository">
    <batch:step id="outputAllCustomerList01.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="reader"
                processor="retrieveBranchFromContextItemProcessor"
                writer="writer" commit-interval="10"/>
            <batch:listeners>
                <batch:listener ref="branchMasterReadStepListener"/> <!-- (6) -->
            </batch:listeners>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

## 一括取得したデータを後続ステップのItemProcessorで参照する例

```
@Component
public class RetrieveBranchFromContextItemProcessor implements
    ItemProcessor<Customer, CustomerWithBranch> {

    private Map<String, Branch> branches;

    @BeforeStep      // (7)
    @SuppressWarnings("unchecked")
    public void beforeStep(StepExecution stepExecution) {
        branches = (Map<String, Branch>) stepExecution.getExecutionContext()
            .get("branches"); // (8)
    }

    @Override
    public CustomerWithBranch process(Customer item) throws Exception {
        CustomerWithBranch newItem = new CustomerWithBranch(item);
        newItem.setBranch(branches.get(item.getChargeBranchId())); // (9)
        return newItem;
    }
}
```

項番	説明
(1)	StepExecutionListenerインターフェースを実装する。 ここでは実装を簡易にするため、StepExecutionListenerインターフェースを実装したStepExecutionListenerSupportからの拡張としている。
(2)	ステップ実行前にデータを取得するため、beforeStepメソッドを実装する。
(3)	マスタデータを取得する処理を実装する。
(4)	後続処理が利用しやすいうようにList型からMap型へ変換を行う。
(5)	ステップのコンテキストに取得したマスタデータをbranchesという名前で設定する。

項目番	説明
(6)	対象となるジョブへ作成したListenerを登録する。
(7)	ItemProcessorのステップ実行前にマスタデータを取得するため、@BeforeStepアノテーションでListener設定を行う。
(8)	@BeforeStepアノテーションが付与されたメソッド内で、ステップのコンテキストから(5)で設定されたマスタデータを取得する。
(9)	ItemProcessorのprocessメソッド内で、マスタデータからデータ取得を行う。



コンテキストへ格納するオブジェクト

コンテキスト(`ExecutionContext`)へ格納するオブジェクトは、`java.io.Serializable`を実装したクラスでなければならない。これは、`ExecutionContext`が`JobRepository`へ格納されるためである。

#### 処理対象レコードに応じて都度取得する場合

業務処理のItemProcessorとは別に、都度取得専用のItemProcessorにて取得する。これにより、各ItemProcessorの処理を簡素化する。

1. 都度取得用のItemProcessorを定義し、業務処理と分離する。
  - この際、テーブルアクセス時はMyBatisをそのまま使う。
2. 複数のItemProcessorをCompositeItemProcessorを使用して連結する。
  - ItemProcessorはdelegates属性に指定した順番に処理されることに留意する。

#### 都度取得用のItemProcessorの実装例

```

@Component
public class RetrieveBranchFromRepositoryItemProcessor implements
    ItemProcessor<Customer, CustomerWithBranch> {

    @Inject
    BranchRepository branchRepository; // (1)

    @Override
    public CustomerWithBranch process(Customer item) throws Exception {
        CustomerWithBranch newItem = new CustomerWithBranch(item);
        newItem.setBranch(branchRepository.findOne(
            item.getChargeBranchId())); // (2)
        return newItem; // (3)
    }
}

```

## 都度取得用と業務処理用のItemProcessorの定義例

```
<bean id="compositeItemProcessor"
      class="org.springframework.batch.item.support.CompositeItemProcessor">
    <property name="delegates">
      <list>
        <ref bean="retrieveBranchFromRepositoryItemProcessor"/> <!-- (4) -->
        <ref bean="businessLogicItemProcessor"/> <!-- (5) -->
      </list>
    </property>
</bean>
```

項番	説明
(1)	MyBatisを利用した都度データ取得用のRepositoryをInjectする。
(2)	入力データ(処理対象レコード)に対して、Repositoryから付随データを取得する。
(3)	処理対象レコードと付随データと一緒にしたデータを返却する。 このデータが次のItemProcessorへの入力データになることに注意する。
(4)	都度取得用のItemProcessorを設定する。
(5)	ビジネスロジックのItemProcessorを設定する。

### 5.1.3.2.2. 複数データソースへの出力(複数ステップ)

データソースごとにステップを分割し、各ステップで單一データソースを処理することで、ジョブ全体で複数データソースを処理する。

- 1ステップ目で加工したデータをテーブルに格納し、2ステップ目でファイルに出力する、といった要領となる。
- 各ステップがシンプルになりリカバリしやすい反面、2度手間になる可能性がある。
  - この結果、以下のような弊害を生む場合は、1ステップで複数データソースを処理することを検討する。
    - 処理時間が伸びてしまう
    - 業務ロジックが冗長となる

### 5.1.3.2.3. 複数データソースへの出力(1ステップ)

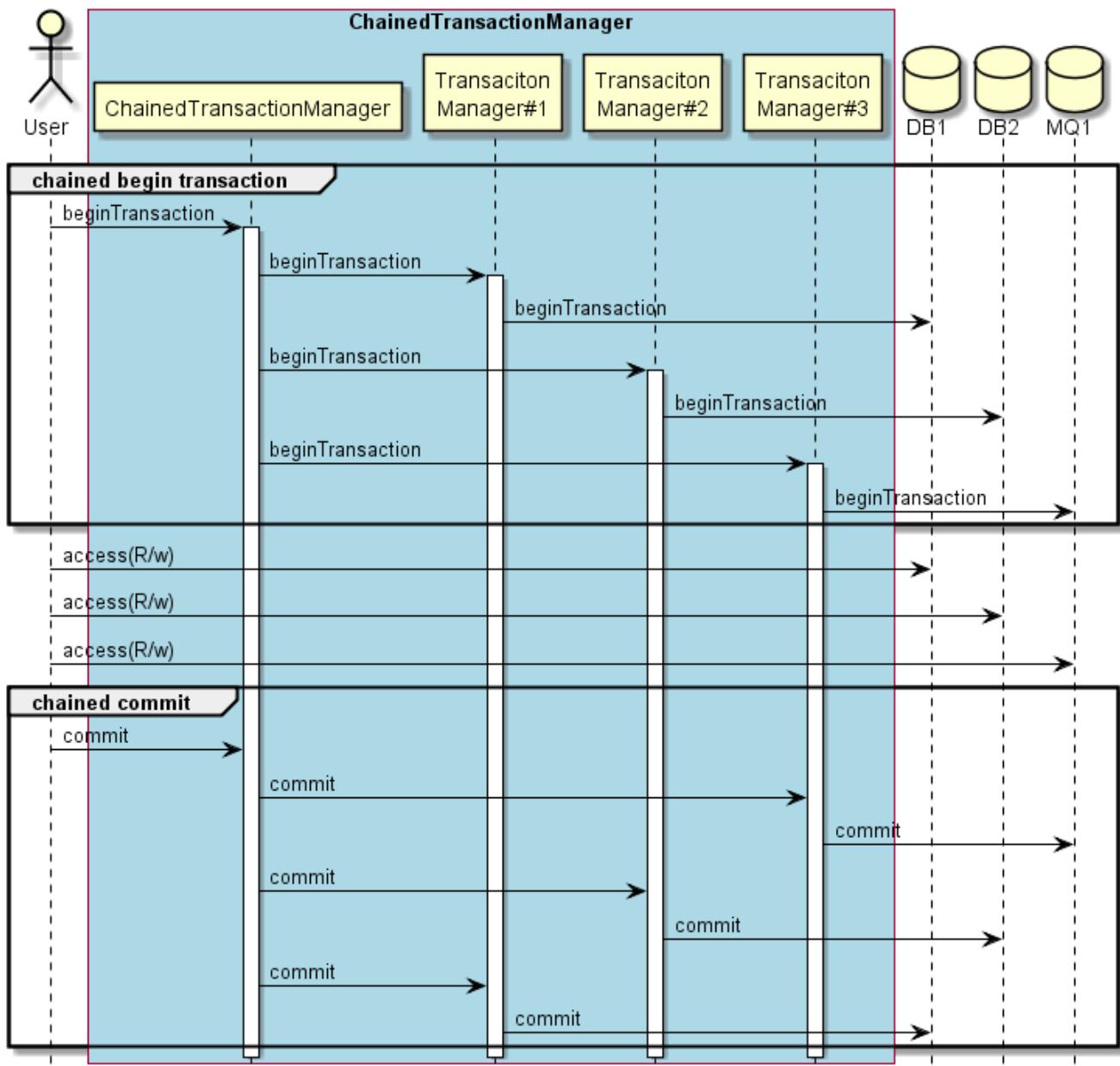
一般的に、複数のデータソースに対するトランザクションを1つにまとめる場合は、2phase-commitによる分散トランザクションを利用する。しかし、以下の様なデメリットがあることも同時に知られている。

- XAResourceなど分散トランザクションAPIにミドルウェアが対応している必要があり、それにもとづいた特殊な設定が必要になる
- バッチプログラムのようなスタンドアロンJavaで、分散トランザクションのJTA実装ライブラリを追加する必要がある
- 障害時のリカバリが難しい

Spring Batchでも分散トランザクションを活用することは可能だが、JTAによるグローバルトランザクションを使用する方法では、プロトコルの特性上、性能面のオーバーヘッドがかかる。より簡易に複数データソースをまとめて処理する方法として、**Best Efforts 1PC/パターン**による実現手段を推奨する。

*Best Efforts 1PC/パターン*とは

端的に言うと、複数データソースをローカルトランザクションで扱い、同じタイミングで逐次コミットを発行する、という手法を指す。下図に概念図を示す。



*Best Efforts 1PC/パターン*の概念図

図の説明

1. ユーザが**ChainedTransactionManager**にトランザクション開始を指示する。
2. **ChainedTransactionManager**は、登録されているトランザクションマネージャを逐次トランザクションを開始する。
3. ユーザは各リソースヘトランザクショナルな操作を行う。
4. ユーザが**ChainedTransactionManager**にコミットを指示する。

5. **ChainedTransactionManager**は、登録されているトランザクションマネージャを逐次コミットを発行する。

- トランザクション開始と逆順にコミット(またはロールバック)される

この方法は分散トランザクションではないため、2番目以降のトランザクションマネージャにおけるcommit/rollback時に障害(例外)が発生した場合に、データの整合性が保てない可能性がある。そのため、トランザクション境界で障害が発生した場合のリカバリ方法を設計する必要があるが、リカバリ頻度を低減し、リカバリ手順を簡素しやすくなる効果がある。

複数のトランザクショナルリソースを同時に処理する場合

複数のデータベースを同時に処理する場合や、データベースとMQを処理する場合などに活用する。

以下のように、**ChainedTransactionManager**を使用して複数トランザクションマネージャを1つにまとめて定義することで1phase-commitとして処理する。なお、**ChainedTransactionManager**はSpring Dataが提供するクラスである。

pom.xml

```
<dependencies>
    <!-- omitted -->
    <!-- (1) -->
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-commons</artifactId>
    </dependency>
</dependencies>
```

## *chainedTransactionManager*の使用例

```
<!-- Chained Transaction Manager -->
<!-- (2) -->
<bean id="chainedTransactionManager"
      class="org.springframework.data.transaction.ChainedTransactionManager">
    <constructor-arg>
      <!-- (3) -->
      <list>
        <ref bean="transactionManager1"/>
        <ref bean="transactionManager2"/>
      </list>
    </constructor-arg>
</bean>

<batch:job id="jobSalesPlan01" job-repository="jobRepository">
  <batch:step id="jobSalesPlan01.step01">
    <!-- (4) -->
    <batch:tasklet transaction-manager="chainedTransactionManager">
      <!-- omitted -->
    </batch:tasklet>
  </batch:step>
</batch:job>
```

項目番	説明
(1)	<i>ChainedTransactionManager</i> を利用するため、依存関係を追加する。
(2)	<i>ChainedTransactionManager</i> のBean定義を行う。
(3)	まとめたい複数のトランザクションマネージャをリストで定義する。
(4)	ジョブが利用するトランザクションマネージャに(1)で定義したBeanIDを指定する。

トランザクショナルリソースと非トランザクショナルリソースを同時に処理する場合

この方法は、データベースとファイルを同時に処理する場合に活用する。

データベースについては[単一データソースの場合](#)と同様。

ファイルについてはFlatFileItemWriterの**transactional**プロパティをtrueに設定することで、前述の「Best Efforts 1PCパターン」と同様の効果となる。

詳細は[非トランザクショナルなデータソースに対する補足](#)を参照。

この設定は、データベースのトランザクションをコミットする直前までファイルへの書き込みを遅延させるため、2つのデータソースで同期がとりやすくなる。ただし、この場合でもデータベースへのコミット後、ファイル出力処理中に異常が発生した場合はデータの整合性が保てない可能性があるため、リカバリ方法を設計する必要がある。

### 5.1.3.3. 中間方式コミットでの注意点

非推奨ではあるがItemWriterで処理データをスキップする場合は、チャンクサイズが設定値か強制変更さ

れる。そのことがトランザクションに非常に大きく影響することに注意する。詳細は、[スキップ](#)を参照。

## 5.2. データベースアクセス

### 5.2.1. Overview

TERASOLUNA Batch 5.xでは、データベースアクセスの方法として、MyBatis3(以降、「MyBatis」と呼ぶ)を利用する。MyBatisによるデータベースアクセスの基本的な利用方法は、TERASOLUNA Server 5.x開発ガイドラインの以下を参照してほしい。

- [データベースアクセス\(共通編\)](#)
- [データベースアクセス\(MyBatis3編\)](#)

本節では、TERASOLUNA Batch 5.x特有の使い方を中心に説明する。

本機能は、チャンクモデルとタスクレットモデルとで使い方が異なるため、それについて説明する。

### 5.2.2. How to use

TERASOLUNA Batch 5.xでのデータベースアクセス方法を説明する。

TERASOLUNA Batch 5.xでのデータベースアクセスは、以下の2つの方法がある。  
これらはデータベースアクセスするコンポーネントによって使い分ける。

1. MyBatis用のItemReaderおよびItemWriterを利用する。
  - チャンクモデルでのデータベースアクセスによる入出力で使用する。
    - org.mybatis.spring.batch.MyBatisCursorItemReader
    - org.mybatis.spring.batch.MyBatisBatchItemWriter
2. Mapperインターフェースを利用する
  - チャンクモデルでのビジネスロジック処理で使用する。
    - ItemProcessor実装で利用する。
  - タスクレットモデルでのデータベースアクセス全般で使用する。
    - Tasklet実装で利用する。

#### 5.2.2.1. 共通設定

データベースアクセスにおいて必要な共通設定について説明を行う。

1. [データソースの設定](#)
2. [MyBatisの設定](#)
3. [Mapper XMLの定義](#)
4. [MyBatis-Springの設定](#)

### 5.2.2.1.1. データソースの設定

TERASOLUNA Batch 5.xでは、2つのデータソースを前提としている。`launch-context.xml`でデフォルト設定している2つのデータソースを示す。

#### データソース一覧

データソース名	説明
<code>adminDataSource</code>	Spring BatchやTERASOLUNA Batch 5.xが利用するデータソース JobRepositoryや <a href="#">非同期実行(DBポーリング)</a> で利用している。
<code>jobDataSource</code>	ジョブが利用するデータソース

以下に、`launch-context.xml`と接続情報のプロパティを示す。

これらをユーザの環境に合わせて設定すること。

`resources|META-INF|spring|launch-context.xml`

```
<!-- (1) -->
<bean id="adminDataSource" class="org.apache.commons.dbcp2.BasicDataSource"
    destroy-method="close"
    p:driverClassName="${admin.jdbc.driver}"
    p:url="${admin.jdbc.url}"
    p:username="${admin.jdbc.username}"
    p:password="${admin.jdbc.password}"
    p:maxTotal="10"
    p:minIdle="1"
    p:maxWaitMillis="5000"
    p:defaultAutoCommit="false"/>

<!-- (2) -->
<bean id="jobDataSource" class="org.apache.commons.dbcp2.BasicDataSource"
    destroy-method="close"
    p:driverClassName="${jdbc.driver}"
    p:url="${jdbc.url}"
    p:username="${jdbc.username}"
    p:password="${jdbc.password}"
    p:maxTotal="10"
    p:minIdle="1"
    p:maxWaitMillis="5000"
    p:defaultAutoCommit="false" />
```

## *batch-application.properties*

```
# (3)
# Admin DataSource settings.
admin.h2.jdbc.driver=org.h2.Driver
admin.h2.jdbc.url=jdbc:h2:mem:batch;DB_CLOSE_DELAY=-1
admin.h2.jdbc.username=sa
admin.h2.jdbc.password=

# (4)
# Job DataSource settings.
jdbc.driver=org.postgresql.Driver
jdbc.url=jdbc:postgresql://localhost:5432/postgres
jdbc.username=postgres
jdbc.password=postgres
```

### 説明

項目番号	説明
(1)	<code>adminDataSource</code> の定義。(3)の接続情報が設定される。
(2)	<code>jobDataSource</code> の定義。(4)の接続情報が設定される。
(3)	<code>adminDataSource</code> で利用するデータベースへの接続情報 この例では、H2を利用している。
(4)	<code>jobDataSource</code> で利用するデータベースへの接続情報 この例では、PostgreSQLを利用している。

### 5.2.2.1.2. MyBatisの設定

TERASOLUNA Batch 5.xで、MyBatisの設定をする上で重要な点について説明をする。

バッチ処理を実装する際の重要なポイントの1つとして「大量のデータを一定のリソースで効率よく処理する」が挙げられる。

これに関する設定を説明する。

- `fetchSize`
  - 一般的なバッチ処理では、大量のデータを処理する際の通信コストを低減するために、JDBCドライバに適切な`fetchSize`を指定することが必須である。`fetchSize`とは、JDBCドライバとデータベース間で1回の通信で取得するデータ件数を設定するパラメータである。この値は出来る限り大きい値を設定することが望ましいが、大きすぎるとメモリを圧迫するため、注意が必要である。ユーザにてチューニングする必要がある箇所と言える。
  - MyBatisでは、全クエリ共通の設定として`defaultFetchSize`を設定することができ、さらにクエリごとの`fetchSize`設定で上書きできる。
- `executorType`
  - 一般的なバッチ処理では、同一トランザクション内で同じSQLを全データ件数/`fetchSize`の回数分実行することになる。この際、都度ステートメントを作成するのではなく再利用することで効率よく処理できる。

- MyBatisの設定における、`defaultExecutorType`に`REUSE`を設定することでステートメントの再利用が可能、処理スループット向上に寄与する。
- 大量のデータを一度に更新する場合、JDBCのバッチ更新を利用することで性能向上が期待できる。  
そのため、`MyBatisBatchItemWriter`で利用する`SqlSessionTemplate`には、`executorType`に(`REUSE`ではなく)`BATCH`が設定されている。

TERASOLUNA Batch 5.xでは、同時に2つの異なる`ExecutorType`が存在する。一方の`ExecutorType`で実装する場合が多いと想定するが、併用時は特に注意が必要である。この点は、`ItemReader`・`ItemWriter`以外のデータベースアクセスにて詳しく説明する。

#### *MyBatisのその他のパラメータ*



その他のパラメータに関しては以下リンクを参照し、アプリケーションの特性にあつた設定を行うこと。

<http://www.mybatis.org/mybatis-3/configuration.html>

以下にデフォルト提供されている設定を示す。

*META-INF/spring/launch-context.xml*

```
<bean id="jobSqlSessionFactory"
      class="org.mybatis.spring.SqlSessionFactoryBean"
      p:dataSource-ref="jobDataSource">
    <!-- (1) -->
    <property name="configuration">
        <bean class="org.apache.ibatis.session.Configuration"
              p:localCacheScope="STATEMENT"
              p:lazyLoadingEnabled="true"
              p:aggressiveLazyLoading="false"
              p:defaultFetchSize="1000"
              p:defaultExecutorType="REUSE"/>
    </property>
</bean>

<!-- (2) -->
<bean id="batchModeSqlSessionTemplate"
      class="org.mybatis.spring.SqlSessionTemplate"
      c:sqlSessionFactory-ref="jobSqlSessionFactory"
      c:executorType="BATCH"/>
```

#### 説明

項番	説明
(1)	MyBatisの各種設定を行う。 デフォルトでは、 <code>fetchSize</code> を1000に設定している。
(2)	<code>MyBatisBatchItemWriter</code> のために、 <code>executorType</code> が <code>BATCH</code> の <code>SqlSessionTemplate</code> を定義している。

*adminDataSource*を利用した*SqlSessionFactory*の定義箇所について

同期実行をする場合は、*adminDataSource*を利用した*SqlSessionFactory*は不要であるため、定義がされていない。[非同期実行\(DBポーリング\)](#)を利用する場合、ジョブ要求テーブルへアクセスするために *META-INF/spring/async-batch-daemon.xml*内に定義されている。

*META-INF/spring/async-batch-daemon.xml*



```
<bean id="adminSqlSessionFactory"
      class="org.mybatis.spring.SqlSessionFactoryBean"
      p:dataSource-ref="adminDataSource" >
    <property name="configuration">
      <bean class="org.apache.ibatis.session.Configuration"
            p:localCacheScope="STATEMENT"
            p:lazyLoadingEnabled="true"
            p:aggressiveLazyLoading="false"
            p:defaultFetchSize="1000"
            p:defaultExecutorType="REUSE"/>
    </property>
</bean>
```

#### 5.2.2.1.3. Mapper XMLの定義

TERASOLUNA Batch 5.x特有の説明事項はないので、TERASOLUNA Server 5.x 開発ガイドラインの[データベースアクセス処理の実装](#)を参照してほしい。

#### 5.2.2.1.4. MyBatis-Springの設定

MyBatis-Springが提供するItemReaderおよびItemWriterを使用する場合、MapperのConfigで使用するMapper XMLを設定する必要がある。

設定方法としては、以下の2つが考えられる。

1. 共通設定として、すべてのジョブで使用するMapper XMLを登録する。
  - *META-INF/spring/launch-context.xml*にすべてのMapper XMLを記述することになる。
2. 個別設定として、ジョブ単位で利用するMapper XMLを登録する。
  - *META-INF/jobs/*配下のBean定義に、個々のジョブごとに必要なMapper XMLを記述することになる。

共通設定をしてしまうと、同期実行をする際に実行するジョブのMapper XMLだけでなく、その他のジョブが使用するMapper XMLも読み込んでしまうために以下に示す弊害が生じる。

- ジョブの起動までに時間がかかる
- メモリリソースの消費が大きくなる

これを回避するために、TERASOLUNA Batch 5.xでは、個別設定として、個々のジョブ定義でそのジョブが必要とするMapper XMLだけを指定する設定方法を採用する。

基本的な設定方法については、TERASOLUNA Server 5.x 開発ガイドラインの [MyBatis-Springの設定](#)を参照してほしい。

TERASOLUNA Batch 5.xでは、複数のSqlSessionFactoryおよびSqlSessionTemplateが定義されているため、どれを利用するか明示的に指定する必要がある。

基本的にはjobSqlSessionFactoryを指定すればよい。

以下に設定例を示す。

META-INF/jobs/common/jobCustomerList01.xml

```
<!-- (1) -->
<mybatis:scan
    base-package="org.terasoluna.batch.functionalttest.app.repository.mst"
    factory-ref="jobSqlSessionFactory"/>
```

#### 説明

項番	説明
(1)	<mybatis:scan>のfactory-ref属性にjobSqlSessionFactoryを設定する。

### 5.2.2.2. ItemReaderにおけるデータベースアクセス

ここではItemReaderによるデータベースアクセスについて説明する。

#### 5.2.2.2.1. MyBatisのItemReader

MyBatis-Springが提供するItemReaderとして下記の2つが存在する。

- org.mybatis.spring.batch.MyBatisCursorItemReader
- org.mybatis.spring.batch.MyBatisPagingItemReader

MyBatisPagingItemReaderは、TERASOLUNA Server 5.x 開発ガイドラインの [Entityのページネーション検索\(SQL絞り込み方式\)](#)で 説明している仕組みを利用したItemReaderである。

一定件数を取得した後に再度SQLを発行するため、データの一貫性が保たれない可能性がある。そのため、バッチ処理で利用するには危険であることから、TERASOLUNA Batch 5.xでは原則使用しない。

TERASOLUNA Batch 5.xではMyBatisCursorItemReaderのみを利用する。

TERASOLUNA Batch 5.xでは、[MyBatis-Springの設定](#)で説明したとおり、mybatis:scanによって動的にMapper XMLを登録する方法を採用している。そのため、Mapper XMLに対応するインターフェースを用意する必要がある。詳細については、TERASOLUNA Server 5.x 開発ガイドラインの [データベースアクセス処理の実装](#)を参照。

MyBatisCursorItemReaderの利用例を以下に示す。

```
<!-- (1) -->
<mybatis:scan
    base-package="org.terasoluna.batch.functionalttest.app.repository.mst"
    factory-ref="jobSqlSessionFactory"/>

<!-- (2) (3) (4) -->
<bean id="reader"
    class="org.mybatis.spring.batch.MyBatisCursorItemReader" scope="step"

p:queryId="org.terasoluna.batch.functionalttest.app.repository.mst.CustomerRepository.
    findAll"
    p:sqlSessionFactory-ref="jobSqlSessionFactory"/>
```

```
<!-- (5) -->
<mapper
namespace="org.terasoluna.batch.functionalttest.app.repository.mst.CustomerRepository">

<!-- (6) -->
<select id="findAll"
    resultType="org.terasoluna.batch.functionalttest.app.model.mst.Customer">
<![CDATA[
    SELECT
        customer_id AS customerId,
        customer_name AS customerName,
        customer_address AS customerAddress,
        customer_tel AS customerTel,
        charge_branch_id AS chargeBranchId,
        create_date AS createDate,
        update_date AS updateDate
    FROM
        customer_mst
    ORDER by
        charge_branch_id ASC, customer_id ASC
    ]]>
</select>

<!-- omitted -->
</mapper>
```

```
public interface CustomerRepository {  
    // (7)  
    List<Customer> findAll();  
  
    // omitted  
}
```

#### 説明

項番	説明
(1)	Mapper XMLの登録を行う。
(2)	<code>MyBatisCursorItemReader</code> を定義する。
(3)	<code>queryId</code> のプロパティに、(6)で定義しているSQLのIDを(5)のnamespace + <メソッド名>で指定する。
(4)	<code>sqlSessionFactory</code> のプロパティに、アクセスするデータベースの <code>SqlSessionFactory</code> を指定する。
(5)	Mapper XMLを定義する。namespaceの値とインターフェースのFQCNを一致させること。
(6)	SQLを定義する。
(7)	(6)で定義したSQLのIDに対応するメソッドをインターフェースに定義する。

#### 5.2.2.3. ItemWriterにおけるデータベースアクセス

ここではItemWriterによるデータベースアクセスについて説明する。

##### 5.2.2.3.1. MyBatisのItemWriter

MyBatis-Springが提供するItemWriterは以下の1つのみである。

- `org.mybatis.spring.batch.MyBatisBatchItemWriter`

基本的な設定については、MyBatisのItemReaderと同じである。MyBatisBatchItemWriterでは、MyBatisの設定で説明したbatchModeSqlSessionTemplateを指定する必要がある。

MyBatisBatchItemWriterの定義例を以下に示す。

*META-INF/jobs/common/jobSalesPlan01.xml*

```
<!-- (1) -->
<mybatis:scan
    base-package="org.terasoluna.batch.functionalttest.app.repository.plan"
    factory-ref="jobSqlSessionFactory"/>

<!-- (2) (3) (4) -->
<bean id="detailWriter" class="org.mybatis.spring.batch.MyBatisBatchItemWriter"
    p:statementId="org.terasoluna.batch.functionalttest.app.repository.plan.SalesPlanDetail
    Repository.create"
    p:sqlSessionTemplate="batchModeSqlSessionTemplate"/>

<!-- omitted -->
```

*org/terasoluna/batch/functionalttest/app/repository/plan/SalesPlanDetailRepository.xml*

```
<!-- (5) -->
<mapper
    namespace="org.terasoluna.batch.functionalttest.app.repository.plan.SalesPlanDetailRepo
    sitory">

    <!-- (6) -->
    <insert id="create"

        parameterType="org.terasoluna.batch.functionalttest.app.model.plan.SalesPlanDetail">
            <![CDATA[
                INSERT INTO
                    sales_plan_detail(branch_id, year, month, customer_id, amount)
                VALUES (
                    #{branchId}, #{year}, #{month}, #{customerId}, #{amount}
                )
            ]]>
    </insert>

    <!-- omitted -->
</mapper>
```

*org.terasoluna.batch.functionalttest.app.repository.plan.SalesPlanDetailRepository*

```
public interface SalesPlanDetailRepository {

    // (7)
    void create(SalesPlanDetail salesPlanDetail);

    // omitted
}
```

## 説明

項目番	説明
(1)	Mapper XMLの登録を行う。
(2)	<code>MyBatisBatchItemWriter</code> を定義する。
(3)	<code>statementId</code> のプロパティに、(6)で定義しているSQLのIDを(5)の <code>namespace + &lt;メソッド名&gt;</code> で指定する。
(4)	<code>sqlSessionTemplate</code> のプロパティに、アクセスするデータベースの <code>SessionTemplate</code> を指定する。 指定する <code>SessionTemplate</code> は、 <code>executorType</code> が <code>BATCH</code> に設定されていることが必須である。
(5)	Mapper XMLを定義する。namespaceの値とインターフェースのFQCNを一致させること。
(6)	SQLを定義する。
(7)	(6)で定義したSQLのIDに対応するメソッドをインターフェースに定義する。

### 5.2.2.4. ItemReader・ItemWriter以外のデータベースアクセス

ItemReader・ItemWriter以外のデータベースアクセスについて説明する。

ItemReader・ItemWriter以外でデータベースアクセスするには、Mapperインターフェースを利用する。Mapperインターフェースを利用するにあたって、TERASOLUNA Batch 5.xでは以下の制約を設けている。

Mapperインターフェースの利用可能な箇所

処理	ItemProcessor	Tasklet	リスナー
参照	利用可	利用可	利用可
更新	条件付で利用可	利用可	利用不可

ItemProcessorでの制約

MyBatisには、同一トランザクション内で2つ以上の`ExecutorType`で実行してはいけないという制約がある。

「ItemWriterに`MyBatisBatchItemWriter`を使用する」と「ItemProcessorでMapperインターフェースを使用し参照更新をする」を同時に満たす場合は、この制約に抵触する。

制約を回避するには、ItemProcessorでは`ExecutorType`が`BATCH`のMapperインターフェースによってデータベースアクセスすることになる。

加えて、`MyBatisBatchItemWriter`ではSQL実行後のステータスチェックにより、自身が発行したSQLかどうかチェックしているのだが、当然ItemProcessorによるSQL実行は管理できないためエラーが発生してしまう。

よって、`MyBatisBatchItemWriter`を利用している場合は、Mapperインターフェースによる更新はできなくなり、参照のみとなる。



`MyBatisBatchItemWriter`のエラーチェックを無効化する設定ができるが、予期せぬ動作が起きる可能性があるため無効化は禁止する。

Taskletでの制約

Taskletでは、Mapperインターフェースを利用することが基本であるため、ItemProcessorのような影

響はない。

`MyBatisBatchItemWriter`をInjectして利用することも考えられるが、その場合はMapperインターフェース自体を `BATCH`設定で処理すればよい。つまり、Taskletでは、`MyBatisBatchItemWriter`をInjectして使う必要は基本的がない。

## リスナーでの制約

リスナーでもItemProcessorでの制約と同じ制約が成立する。加えて、リスナーでは、更新を必要とするユースケースを考えにくい。よって、リスナーでは、更新系処理を禁止する。

リスナーで想定される更新処理の代替

ジョブの状態管理



Spring BatchのJobRepositoryによって行われている

データベースへのログ出力

ログのAppenderで実施すべき。ジョブのトランザクションとも別管理する必要がある。

### 5.2.2.4.1. ItemProcessorでのデータベースアクセス

ItemProcessorでのデータベースアクセス例を説明する。

*ItemProcessor*での実装例

```
@Component
public class UpdateItemFromDBProcessor implements
    ItemProcessor<SalesPerformanceDetail, SalesPlanDetail> {

    // (1)
    @Inject
    CustomerRepository customerRepository;

    @Override
    public SalesPlanDetail process(SalesPerformanceDetail readItem) throws Exception {

        // (2)
        Customer customer = customerRepository.findOne(readItem.getCustomerId());

        // (3)
        SalesPlanDetail writeItem = new SalesPlanDetail();
        writeItem.setBranchId(customer.getChargeBranchId());
        writeItem.setYear(readItem.getYear());
        writeItem.setMonth(readItem.getMonth());
        writeItem.setCustomerId(readItem.getCustomerId());
        writeItem.setAmount(readItem.getAmount());
        return writeItem;
    }
}
```

## Bean定義

```
<!-- (2) -->
<mybatis:scan
    base-package="org.terasoluna.batch.functionalttest.app.repository"
    template-ref="batchModeSqlSessionTemplate"/>

<bean id="reader" class="org.mybatis.spring.batch.MyBatisCursorItemReader"
    p:queryId="org.terasoluna.batch.functionalttest.app.repository.performance.SalesPerformanceDetailRepository.findAll"
    p:sqlSessionFactory-ref="jobSqlSessionFactory"/>

<!-- (3) -->
<bean id="writer" class="org.mybatis.spring.batch.MyBatisBatchItemWriter"
    p:statementId="org.terasoluna.batch.functionalttest.app.repository.plan.SalesPlanDetailRepository.create"
    p:sqlSessionTemplate-ref="batchModeSqlSessionTemplate"/>

<batch:job id="DBAccessByItemProcessor" job-repository="jobRepository">
    <batch:step id="DBAccessByItemProcessor.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <!-- (4) -->
            <batch:chunk reader="reader"
                processor="updateItemFromDBProcessor"
                writer="writer" commit-interval="10"/>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

MapperインターフェースとMapper XMLは省略する。

### 説明

項番	説明
(1)	MapperインターフェースをInjectする。
(2)	Mapper XMLの登録を行う。 <code>template-ref</code> 属性にBATCH設定されている <code>batchModeSqlSessionTemplate</code> を指定することで、 ItemProcessorでのデータベースアクセスはBATCHとなる。ここで、 <code>factory-ref="jobSqlSessionFactory"</code> としてしまうと、前述の制約に抵触し、 <code>MyBatisBatchItemWriter</code> 実行時に例外が発生してしまう。
(3)	<code>MyBatisBatchItemWriter</code> を定義する。 <code>sqlSessionTemplate</code> プロパティにBATCH設定されている <code>batchModeSqlSessionTemplate</code> を指定する。
(4)	MapperインターフェースをInjectしたItemProcessorを設定する。

### *MyBatisCursorItemReader*設定の補足

以下に示す定義例のように、*MyBatisCursorItemReader*と*MyBatisBatchItemWriter*で異なる*ExecutorType*を使用しても問題ない。これは、*MyBatisCursorItemReader*によるリソースのオープンが、トランザクション開始前に行われているからである。



```
<bean id="reader"
  class="org.mybatis.spring.batch.MyBatisCursorItemReader"
  p:queryId="xxx"
  p:sqlSessionFactory-ref="jobSqlSessionFactory"/>

<bean id="writer"
  class="org.mybatis.spring.batch.MyBatisBatchItemWriter"
  p:statementId="yyy"
  p:sqlSessionTemplate-ref="batchModeSqlSessionTemplate"/>
```

#### 5.2.2.4.2. Taskletでのデータベースアクセス

Taskletでのデータベースアクセス例を説明する。

## Taskletでの実装例

```
@Component
public class OptimisticLockTasklet implements Tasklet {

    // (1)
    @Inject
    ExclusiveControlRepository repository;

    // omitted

    @Override
    public RepeatStatus execute(StepContribution contribution,
        ChunkContext chunkContext) throws Exception {

        Branch branch = repository.branchFindOne(branchId); // (2)
        ExclusiveBranch exclusiveBranch = new ExclusiveBranch();

        exclusiveBranch.setBranchId(branch.getBranchId());
        exclusiveBranch.setBranchName(branch.getBranchName() + " - " + identifier);
        exclusiveBranch.setBranchAddress(branch.getBranchAddress() + " - " +
identifier);
        exclusiveBranch.setBranchTel(branch.getBranchTel());
        exclusiveBranch.setCreateDate(branch.getUpdateDate());
        exclusiveBranch.setUpdateDate(new Timestamp(System.currentTimeMillis()));
        exclusiveBranch.setOldBranchName(branch.getBranchName());

        int result = repository.branchExclusiveUpdate(exclusiveBranch); // (3)

        return RepeatStatus.FINISHED;
    }
}
```

## Bean定義

```
<!-- (4) -->
<mybatis:scan
    base-
    package="org.terasoluna.batch.functionaltest.ch05.exclusivecontrol.repository"
    factory-ref="jobSqlSessionFactory"/>

<batch:job id="taskletOptimisticLockCheckJob" job-repository="jobRepository">
    <batch:step id="taskletOptimisticLockCheckJob.step01">
        <batch:tasklet transaction-manager="jobTransactionManager"
            ref="optimisticLockTasklet"> <!-- (5) -->
            </batch:tasklet>
        </batch:step>
    </batch:job>
```

MapperインターフェースとMapper XMLは省略する。

## 説明

項目番号	説明
(1)	MapperインターフェースをInjectする。
(2)	Mapperインターフェースで検索処理を実行する。
(3)	Mapperインターフェースで更新処理を実行する。
(4)	Mapper XMLの登録を行う。 <code>factory-ref</code> 属性にREUSE設定されている <code>jobSqlSessionFactory</code> を指定する。
(5)	MapperインターフェースをInjectしTaskletを設定する。

### *batchModeSqlSessionTemplate*の利用



タスクレットモデルでの更新処理が多い場合は、`factory-ref`属性

に`'batchModeSqlSessionTemplate`を設定する。これにより、バッチ更新処理が行われるので、性能向上が期待できる。ただし、バッチ更新の実行は`flush`を明示的に呼ぶ必要があるため、注意すること。詳細は、[「バッチモードのRepository利用時の注意点」](#)を参照のこと。

#### 5.2.2.4.3. リスナーでのデータベースアクセス

リスナーでのデータベースアクセスは他のコンポーネントと連携することが多い。使用するリスナー及び実装方法によっては、Mapperインターフェースで取得したデータを、他のコンポーネントへ引き渡す仕組みを追加で用意する必要がある。

ここでは一例として、`StepExecutionListener`でステップ実行前にデータを取得して、`ItemProcessor`で取得したデータを利用する例を示す。

##### リスナーでの実装例

```
public class CacheSetListener extends StepExecutionListenerSupport {

    // (1)
    @Inject
    CustomerRepository customerRepository;

    // (2)
    @Inject
    CustomerCache cache;

    @Override
    public void beforeStep(StepExecution stepExecution) {
        // (3)
        customerRepository.findAll().forEach(customer ->
            cache.addCustomer(customer.getId(), customer));
    }
}
```

## ItemProcessorでの利用例

```
@Component
public class UpdateItemFromCacheProcessor implements
    ItemProcessor<SalesPerformanceDetail, SalesPlanDetail> {

    // (4)
    @Inject
    CustomerCache cache;

    @Override
    public SalesPlanDetail process(SalesPerformanceDetail readItem) throws Exception {
        Customer customer = cache.getCustomer(readItem.getCustomerId()); // (5)

        SalesPlanDetail writeItem = new SalesPlanDetail();

        // omitted
        writerItem.setCustomerName(customer.getCustomerName()); // (6)

        return writeItem;
    }
}
```

## キャッシュクラス

```
// (7)
@Component
public class CustomerCache {

    Map<String, Customer> customerMap = new HashMap<>();

    public Customer getCustomer(String customerId) {
        return customerMap.get(customerId);
    }

    public void addCustomer(String id, Customer customer) {
        customerMap.put(id, customer);
    }
}
```

## Bean定義

```
<!-- omitted -->

<!-- (8) -->
<mybatis:scan
    base-package="org.terasoluna.batch.functionalttest.app.repository"
    template-ref="batchModeSqlSessionTemplate"/>

<!-- (9) -->
<bean id="cacheSetListener"
    class="org.terasoluna.batch.functionalttest.ch05.dbaccess.CacheSetListener"/>

<!-- omitted -->

<batch:job id="DBAccessByItemListener" job-repository="jobRepository">
    <batch:step id="DBAccessByItemListener.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="reader"
                processor="updateItemFromCacheProcessor"
                writer="writer" commit-interval="10"/> <!-- (10) -->
            <!-- (11) -->
            <batch:listeners>
                <batch:listener ref="cacheSetListener"/>
            </batch:listeners>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

## 説明

項目番号	説明
(1)	MapperインターフェースをInjectする。
(2)	Mapperインターフェースから取得したデータをキャッシュするためのBeanをInjectする。
(3)	リスナーにて、Mapperインターフェースからデータを取得してキャッシュする。 ここでは、 <a href="#">StepExecutionListener#beforeStep</a> にてステップ実行前にキャッシュを作成し、以降の処理ではキャッシュを参照することで、I/Oを低減し処理効率を高めている。
(4)	(2)で設定したキャッシュと同じBeanをInjectする。
(5)	キャッシュから該当するデータを取得する。
(6)	更新データにキャッシュからのデータを反映する。
(7)	キャッシュクラスをコンポーネントとして実装する。 ここではBeanスコープは <a href="#">singleton</a> にしている。ジョブに応じて設定すること。
(8)	Mapper XMLの登録を行う。 <a href="#">template-ref</a> 属性に <a href="#">BATCH</a> が設定されている <a href="#">batchModeSqlSessionTemplate</a> を指定する。
(9)	Mapperインターフェースを利用するリスナーを定義する。
(10)	キャッシュを利用するItemProcessorを指定する。

(11)

(9)で定義したリスナーを登録する。

リスナーでの`SqlSessionFactory`の利用

上記の例では、`batchModeSqlSessionTemplate`を設定している  
が、`jobSqlSessionFactory`を設定してもよい。



チャックのスコープ外で動作するリスナーについては、トランザクション外で処理さ  
れるため、`jobSqlSessionFactory`を設定しても問題ない。

## 5.3. ファイルアクセス

### 5.3.1. Overview

本節では、ファイルの入出力を行う方法について説明する。

本機能は、チャンクモデルとタスクレットモデルとで同じ使い方になる。

#### 5.3.1.1. 扱えるファイルの種類

##### 扱えるファイルの種類

TERASOLUNA Batch 5.xで扱えるファイルは以下のとおりである。

これは、Spring Batchにて扱えるものと同じである。

- フラットファイル
- XML

ここではフラットファイルの入出力を行うための方法について説明したのち、 XMLについて[How To Extend](#)で説明する。

まず、TERASOLUNA Batch 5.xで扱えるフラットファイルの種類を示す。

フラットファイルにおける行をここでは**レコード**と呼び、 ファイルの種類はレコードの形式にもとづく、 とする。

##### レコード形式

形式	概要
可変長レコード	CSVやTSVに代表される区切り文字により各項目を区切ったレコード形式。各項目の長さが可変である。
固定長レコード	項目の長さ(バイト数)により各項目を区切ったレコード形式。各項目の長さが固定である。
单一文字列レコード	1レコードを1文字列として扱う形式。

##### 扱えるファイルの構造

フラットファイルの基本構造は以下の2点から構成される。

- レコード区分
- レコードフォーマット

##### フラットファイルのフォーマットを構成する要素

要素	概要
レコード区分	レコードの種類、役割を指す。ヘッダ、データ、トレーラなどがある。詳しくは後述する。
レコードフォーマット	ヘッダ、データ、トレーラレコードがそれぞれ何行あるのか、ヘッダ部～トレーラ部が複数回繰り返されるかなど、レコードの構造を指す。シングルフォーマットとマルチフォーマットがある。詳しくは後述する。

TERASOLUNA Batch 5.xでは、各種レコード区分をもつシングルフォーマットおよびマルチフォーマットのフラットファイルを扱うことができる。

各種レコード区分およびレコードフォーマットについて説明する。

各種レコード区分の概要を以下に示す。

#### レコード区分ごとの特徴

レコード区分	概要
ヘッダレコード	ファイル(データ部)の先頭に付与されるレコードである。 フィールド名、ファイル共通の事項、データ部の集計情報などをもつ。
データレコード	ファイルの主な処理対象となるデータをもつレコードである。
トレーラ/フッタレコード	ファイル(データ部)の末尾に付与されるレコードである。 ファイル共通の事項、データ部の集計情報などをもつ。 シングルフォーマットの場合、フッタレコードと呼ばれることがある。
フッタ/エンドレコード	マルチフォーマットの場合にファイルの末尾に付与されるレコードである。 ファイル共通の事項、ファイル全体の集計情報などをもつ。

##### レコード区分を示すフィールドについて

ヘッダレコードやトレーラレコードをもつフラットファイルでは、レコード区分を示すフィールドをもたせる場合がある。



TERASOLUNA Batch 5.xでは特にマルチフォーマットファイルの処理において、レコード区分ごとに異なる処理を実施する場合などにレコード区分のフィールドを活用する。

レコード区分によって実行する処理を選択する場合の実装は、[マルチフォーマット](#)を参考すること。

##### ファイルフォーマット関連の名称について



個々のシステムにおけるファイルフォーマットの定義によっては、フッタレコードをエンドレコードと呼ぶなど等ガイドラインとは異なる名称が使われている場合がある。

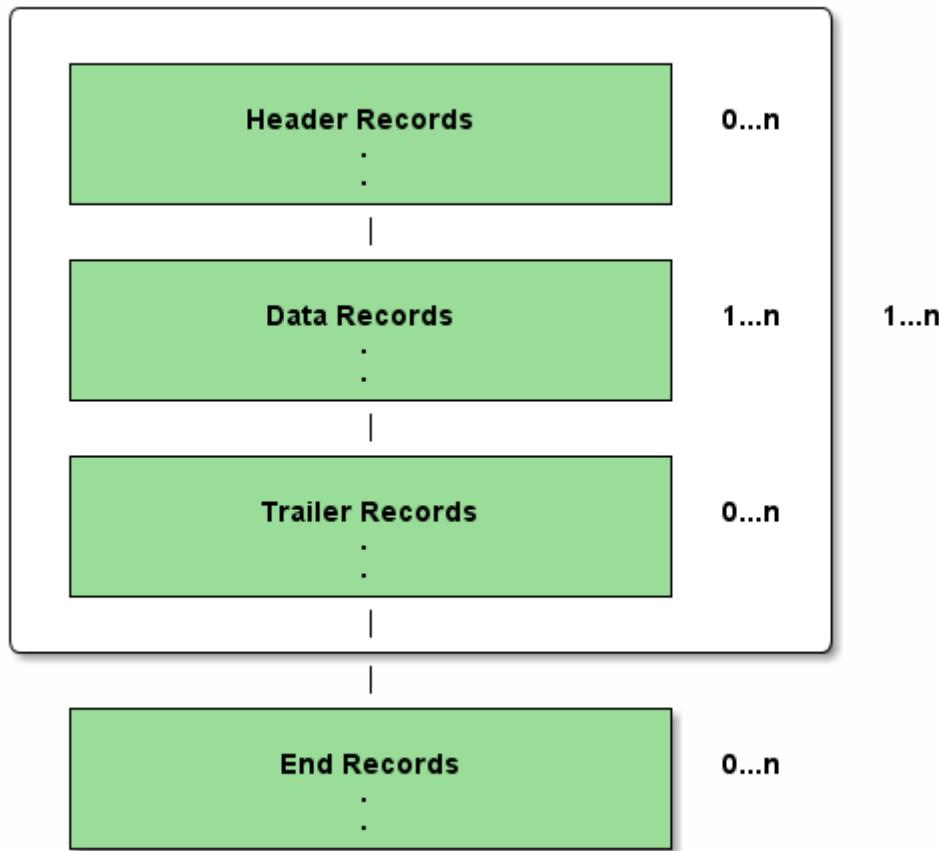
適宜読み替えを行うこと。

シングルフォーマットおよびマルチフォーマットの概要を以下に示す。

#### シングルフォーマットおよびマルチフォーマットの概要

フォーマット	概要
シングルフォーマット	ヘッダn行 + データn行 + トレーラn行 の形式である。
マルチフォーマット	(ヘッダn行 + データn行 + トレーラn行)* n + フッタn行 の形式である。 シングルフォーマットを複数回繰り返した後にフッタレコードが付与されている形式である。

マルチフォーマットのレコード構成を図に表すと下記のようになる。



### マルチフォーマットのレコード構成図

シングルフォーマット、マルチフォーマットフラットファイルの例を以下に示す。  
なお、ファイルの内容説明に用いるコメントアウトを示す文字として//を使用する。

### シングルフォーマット、レコード区分なしフラットファイル(CSV形式)の例

```

branchId,year,month,customerId,amount // (1)
000001,2016,1,0000000001,100000000 // (2)
000001,2016,1,0000000002,200000000 // (2)
000001,2016,1,0000000003,300000000 // (2)
000001,3,600000000 // (3)
    
```

### ファイルの内容の項目一覧

項目番	説明
(1)	ヘッダレコードである。 データ部のフィールド名を示している。
(2)	データレコードである。
(3)	トレーラレコードである。 データ部の集計情報を保持している。

## シングルフォーマット、レコード区分ありのフラットファイル(CSV形式)の例

```
// (1)
H,branchId,year,month,customerId,amount // (2)
D,000001,2016,1,0000000001,100000000
D,000001,2016,1,0000000002,200000000
D,000001,2016,1,0000000003,300000000
T,000001,3,600000000
H,branchId,year,month,customerId,amount // (2)
D,00002,2016,1,0000000004,400000000
D,00002,2016,1,0000000005,500000000
D,00002,2016,1,0000000006,600000000
T,00002,3,1500000000
H,branchId,year,month,customerId,amount // (2)
D,00003,2016,1,0000000007,700000000
D,00003,2016,1,0000000008,800000000
D,00003,2016,1,0000000009,900000000
T,00003,3,2400000000
F,3,9,4500000000 // (3)
```

### ファイルの内容の項目一覧

項目番	説明
(1)	レコードの先頭にレコード区分を示すフィールドをもっている。 それぞれ下記のレコード区分を示す。 <b>H</b> : ヘッダレコード <b>D</b> : データレコード <b>T</b> : トレーラレコード <b>F</b> : フッタレコード
(2)	branchIdが変わることにヘッダ、データ、トレーラを3回繰り返している。
(3)	フッタレコードである。 ファイル全体の集計情報を保持している。

#### データ部のフォーマットに関する前提



[How to use](#)では、データ部のレイアウトは同一のフォーマットである事を前提して説明する。

これは、データ部のレコードはすべて同じ変換対象クラスへマッピングされることを意味する。

#### マルチフォーマットファイルの説明について



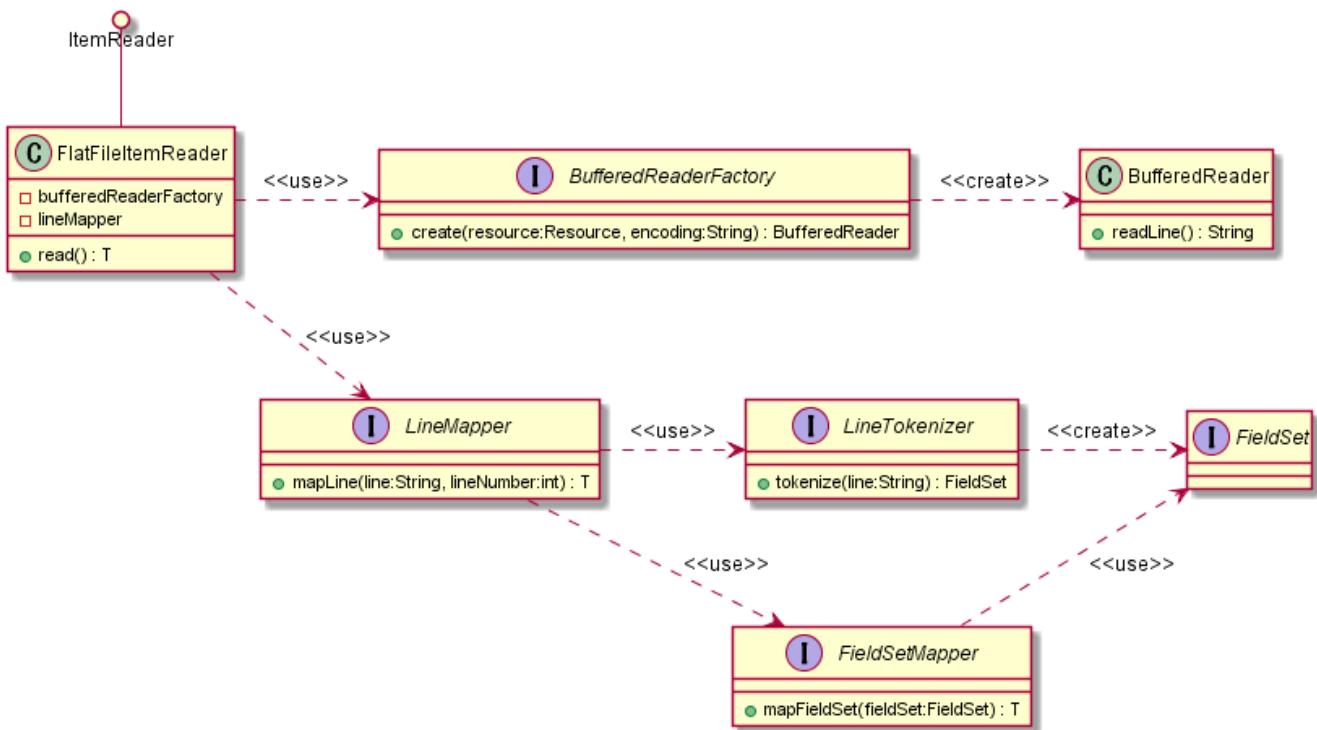
- [How to use](#)では、シングルフォーマットファイルについて説明する。
- マルチフォーマットや上記の構造にフッタ部を含む構造をもつフラットファイルについては、[How To Extend](#)を参照すること

### 5.3.1.2. フラットファイルの入出力を行うコンポーネント

フラットファイルを扱うためのクラスを示す。

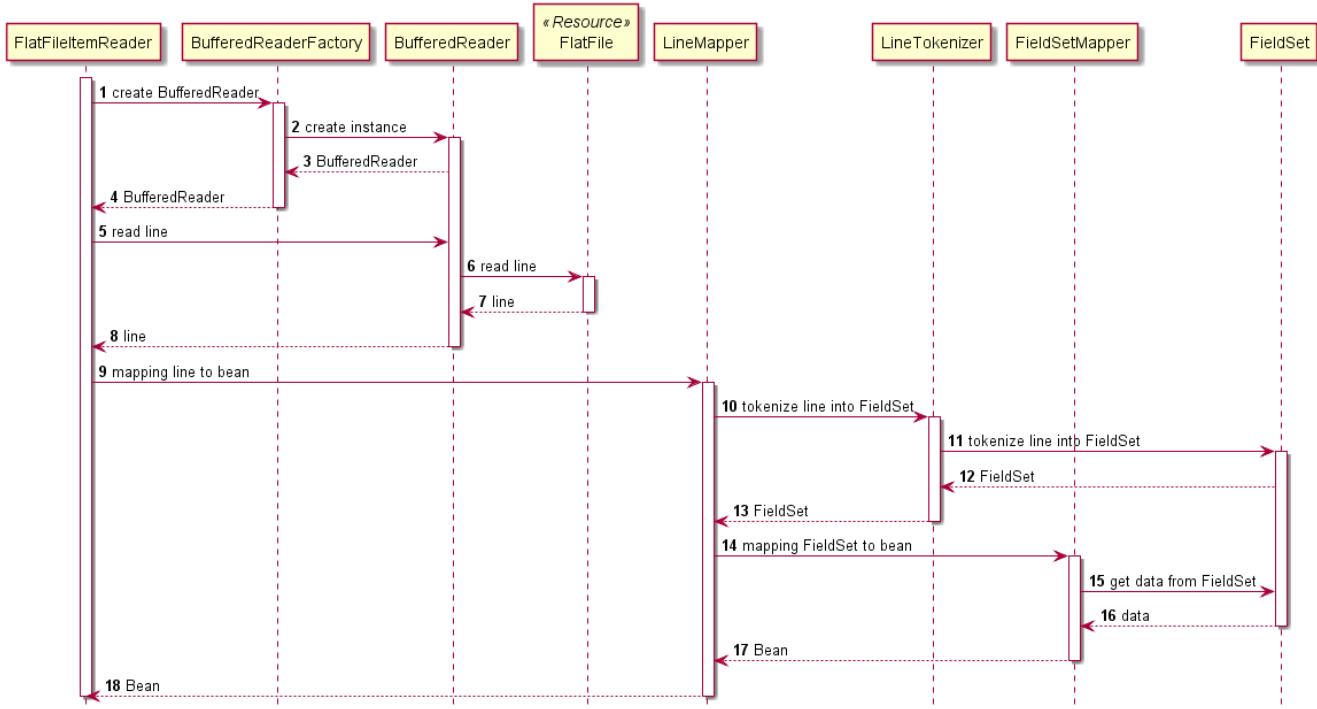
## 入力

フラットファイルの入力を行うために使用するクラスの関連は以下のとおりである。



フラットファイルの入力を行うために使用するクラスの関連

各コンポーネントの呼び出し関係は以下のとおりである。



各コンポーネントの呼び出し関係

各コンポーネントの詳細を以下に示す。

`org.springframework.batch.item.file.FlatFileItemReader`

フラットファイルを読み込みに使用する `ItemReader` の実装クラス。以下のコンポーネントを利用する。

簡単な処理の流れは以下のとおり。

1. `BufferedReaderFactory`を使用して`BufferedReader`を取得する。
2. 取得した`BufferedReader`を使用してフラットファイルから1レコードを読み込む。
3. `LineMapper`を使用して1レコードを対象Beanへマッピングする。

`org.springframework.batch.item.file.BufferedReaderFactory`

ファイルを読み込むための`BufferedReader`を生成する。

`org.springframework.batch.item.file.LineMapper`

1レコードを対象Beanへマッピングする。以下のコンポーネントを利用する。

簡単な処理の流れは以下のとおり。

1. `LineTokenizer`を使用して1レコードを各項目に分割する。
2. `FieldSetMapper`によって分割した項目をBeanのプロパティにマッピングする。

`org.springframework.batch.item.file.transform.LineTokenizer`

ファイルから取得した1レコードを各項目に分割する。

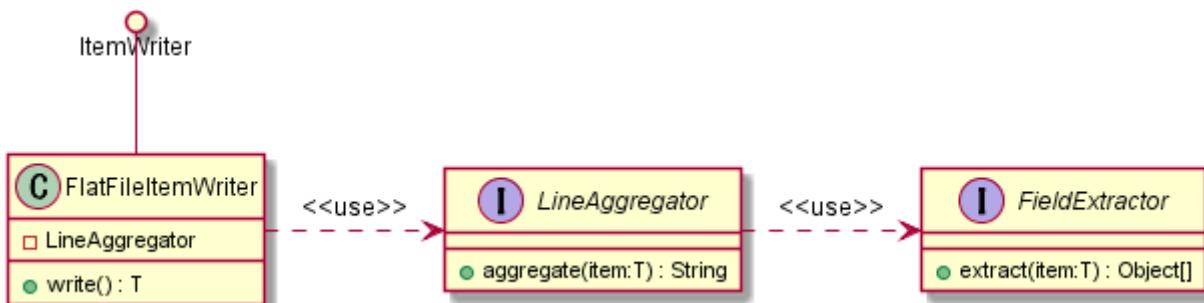
分割された各項目は`FieldSet`クラスに格納される。

`org.springframework.batch.item.file.mapping.FieldSetMapper`

分割した1レコード内の各項目を対象Beanのプロパティへマッピングする。

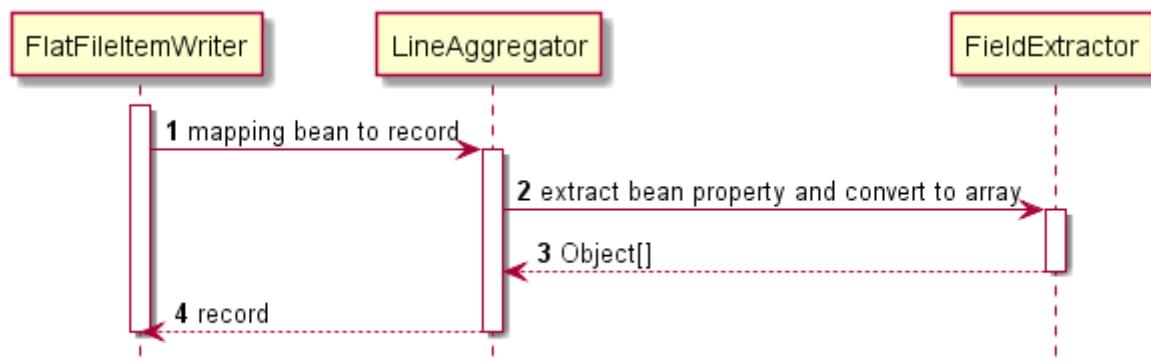
## 出力

フラットファイルの出力を行うために使用するクラスの関連は以下のとおりである。



フラットファイルの出力を行うために使用するクラスの関連

各コンポーネントの呼び出し関係は以下のとおりである。



各コンポーネントの呼び出し関係

`org.springframework.batch.item.file.FlatFileItemWriter`

フラットファイルへの書き出しに使用する `ItemWriter` の実装クラス。以下のコンポーネントを利用する。  
。 `LineAggregator` 対象Beanを1レコードへマッピングする。

`org.springframework.batch.item.file.transform.LineAggregator`

対象Beanを1レコードへマッピングするために使う。 Beanのプロパティとレコード内の各項目とのマッピングは `FieldExtractor` で行う。

`org.springframework.batch.item.file.transform.FieldExtractor`

対象Beanのプロパティを1レコード内の各項目へマッピングする。

### 5.3.2. How to use

フラットファイルのレコード形式別に使い方を説明する。

- 可変長レコード
- 固定長レコード
- 単一文字列レコード

その後、以下の項目について説明する。

- ヘッダとフッタ
- 複数ファイル
- コントロールブレイク

#### 5.3.2.1. 可変長レコード

可変長レコードファイルを扱う場合の定義方法を説明する。

##### 5.3.2.1.1. 入力

下記の入力ファイルを読み込むための設定例を示す。

入力ファイル例

```
000001,2016,1,0000000001,1000000000  
000002,2017,2,0000000002,2000000000  
000003,2018,3,0000000003,3000000000
```

## 変換対象クラス

```
public class SalesPlanDetail {  
  
    private String branchId;  
    private int year;  
    private int month;  
    private String customerId;  
    private BigDecimal amount;  
  
    // omitted getter/setter  
}
```

上記のファイルを読む込むための設定は以下のとおり。

### Bean定義例

```
<!-- (1) (2) (3) -->  
<bean id="reader"  
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"  
      p:resource="#{jobParameters[inputFile]}"  
      p:encoding="MS932"  
      p:strict="true">  
    <property name="lineMapper"> <!-- (4) -->  
      <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">  
        <property name="lineTokenizer"> <!-- (5) -->  
          <!-- (6) (7) (8) -->  
          <bean  
            class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"  
            p:names="branchId,year,month,customerId,amount"  
            p:delimiter=","  
            p:quoteCharacter='''/>  
        </property>  
        <property name="fieldSetMapper"> <!-- (9) -->  
          <bean  
            class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"  
            p:targetType="org.terasoluna.batch.functionalttest.app.model.plan.SalesPlanDetail"/>  
        </property>  
      </bean>  
    </property>  
</bean>
```

### 設定内容の項目一覧

項目番	プロパティ名	設定内容	必須	デフォルト値
(1)	resource	入力ファイルを設定する。	✓	なし

項目番	プロパティ名	設定内容	必須	デフォルト値
(2)	encoding	入力ファイルの文字コードを設定する。		JavaVMのデフォルト文字セット
(3)	strict	trueを設定すると、入力ファイルが存在しない(開けない)場合に例外が発生する。		true
(4)	lineMapper	<code>org.springframework.batch.item.file.mapping.DefaultLineMapper</code> を設定する。 <code>DefaultLineMapper</code> は、設定された <code>LineTokenizer</code> と <code>FieldSetMapper</code> を用いてレコードを変換対象クラスへ変換する基本的な動作を提供する <code>LineMapper</code> である。	✓	なし
(5)	lineTokenizer	<code>org.springframework.batch.item.file.transform.DelimitedLineTokenizer</code> を設定する。 <code>DelimitedLineTokenizer</code> は、区切り文字を指定してレコードを分割する <code>LineTokenizer</code> の実装クラス。 CSV形式の一般的書式とされるRFC-4180の仕様に定義されている、エスケープされた改行、区切り文字、囲み文字の読み込みに対応している。	✓	なし
(6)	names	1レコードの各項目に名前を付与する。 <code>FieldSetMapper</code> で使われる <code>FieldSet</code> で設定した名前を用いて各項目を取り出すことができるようになる。 レコードの先頭から各名前をカンマ区切りで設定する。 <code>BeanWrapperFieldSetMapper</code> を利用する場合は、必須設定である。		なし
(7)	delimiter	区切り文字を設定する		カンマ
(8)	quoteCharacter	囲み文字を設定する		なし
(9)	fieldSetMapper	文字列や数字など特別な変換処理が不要な場合は、 <code>org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper</code> を利用し、プロパティ <code>targetType</code> に変換対象クラスを指定する。 これにより、(5)で設定した各項目の名前と一致するフィールドに値を自動的に設定したインスタンスを生成する。 変換処理が必要な場合は、 <code>org.springframework.batch.item.file.mapping.FieldSetMapper</code> の実装クラスを設定する。	✓	なし



#### *FieldSetMapper*の独自実装について

*FieldSetMapper*を独自に実装する場合については、[How To Extend](#)を参照すること。

### TSV形式ファイルの入力方法

TSVファイルの読み込みを行う場合には、区切り文字にタブを設定することで実現可能である。

#### TSVファイル読み込み時:区切り文字設定例(定数による設定)



```
<property name="delimiter">
  <util:constant
    static-
    field="org.springframework.batch.item.file.transform.DelimitedLineToke
nizer.DELIMITER_TAB"/>
</property>
```

または、以下のようにしてもよい。

#### TSVファイル読み込み時:区切り文字設定例(文字参照による設定)

```
<property name="delimiter" value="&#09;"/>
```

### 5.3.2.1.2. 出力

下記の出力ファイルを書き出すための設定例を示す。

#### 出力ファイル例

```
001,CustomerName001,CustomerAddress001,1111111111,001
002,CustomerName002,CustomerAddress002,1111111111,002
003,CustomerName003,CustomerAddress003,1111111111,003
```

#### 変換対象クラス

```
public class Customer {

  private String customerId;
  private String customerName;
  private String customerAddress;
  private String customerTel;
  private String chargeBranchId;
  private Timestamp createDate;
  private Timestamp updateDate;

  // omitted getter/setter
}
```

上記のファイルを書き出すための設定は以下のとおり。

## Bean定義例

```
<!-- Writer -->
<!-- (1) (2) (3) (4) (5) (6) (7) -->
<bean id="writer"
    class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step"
    p:resource="file:#{jobParameters[outputFile]}"
    p:encoding="MS932"
    p:lineSeparator="\n"
    p:appendAllowed="true"
    p:shouldDeleteIfEmpty="true"
    p:shouldDeleteIfExists="false"
    p:transactional="true">
    <property name="lineAggregator"> <!-- (8) -->
        <bean
            class="org.springframework.batch.item.file.transform.DelimitedLineAggregator"
            p:delimiter=","> <!-- (9) -->
            <property name="fieldExtractor"> <!-- (10) -->
                <!-- (11) -->
                <bean
                    class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"
                    p:names="customerId,(customerName, customerAddress, customerTel, chargeBranchId)">
                    </property>
                </bean>
            </property>
        </bean>
    </property>
</bean>
```

## 設定内容の項目一覧

項目番	プロパティ名	設定内容	必須	デフォルト値
(1)	resource	出力ファイルを設定する。	✓	なし
(2)	encoding	入力ファイルの文字コードを設定する。		JavaVMのデフォルト文字セット
(3)	lineSeparator	レコード区切り(改行コード)を設定する。		システムプロパティのline.separator
(4)	appendAllowed	trueの場合、既存のファイルに追記をする。		false
(5)	shouldDeleteIfEmpty	trueの場合、出力結果が空ファイルであれば削除する。		false
(6)	shouldDeleteIfExists	trueの場合、既にファイルが存在すれば削除する。 falseの場合、既にファイルが存在すれば例外をスローする。		true
(7)	transactional	トランザクション制御を行うかを設定する。 詳細は、 <a href="#">トランザクション制御</a> を参照のこと。		true

項目番	プロパティ名	設定内容	必須	デフォルト値
(8)	lineAggregator	org.springframework.batch.item.file.transform.DelimitedLineAggregatorを設定する。 フィールドを囲み文字で囲む場合は、org.terasoluna.batch.item.file.transform.EnclosableDelimitedLineAggregatorを設定する。 EnclosableDelimitedLineAggregatorの使用方法は後述する。	✓	なし
(9)	delimiter	区切り文字を設定する。		カンマ
(10)	fieldExtractor	文字列や数字など特別な変換処理が不要な場合は、org.springframework.batch.item.file.transform.BeanWrapperFieldExtractorが利用できる。 変換処理が必要な場合は、org.springframework.batch.item.file.transform.FieldExtractorの実装クラスを設定する。 FieldExtractorの実装例は固定長ファイルの出力にて、全角文字のフォーマットを例に説明しているためそちらを参照すること。	✓	なし
(11)	names	1レコードの各項目に名前を付与する。 レコードの先頭から各名前をカンマ区切りで設定する。	✓	なし

#### EnclosableDelimitedLineAggregatorの使用方法

フィールドを囲み文字で囲む場合は、TERASOLUNA Batch 5.xが提供するorg.terasoluna.batch.item.file.transform.EnclosableDelimitedLineAggregatorを使用する。EnclosableDelimitedLineAggregatorの仕様は以下のとおり。

- 囲み文字、区切り文字を任意に指定可能
  - デフォルトはCSV形式で一般的に使用される以下の値である
    - 囲み文字 : "(ダブルクオート)
    - 区切り文字 : ,(カンマ)
- フィールドに行頭復帰、改行、囲み文字、区切り文字が含まれている場合、囲み文字でフィールドを囲む
  - 囲み文字が含まれている場合、直前に囲み文字を付与しエスケープする
  - 設定によってすべてのフィールドを囲み文字で囲むことが可能

EnclosableDelimitedLineAggregatorの使用方法を以下に示す。

#### 出力ファイル例

```
"001","CustomerName""001""","CustomerAddress,001","111111111111","001"
"002","CustomerName""002""","CustomerAddress,002","111111111111","002"
"003","CustomerName""003""","CustomerAddress,003","111111111111","003"
```

## 変換対象クラス

```
// 上記の例と同様
```

Bean定義例(*lineAggregator*の設定のみ)

```
<property name="lineAggregator"> <!-- (1) -->
  <!-- (2) (3) (4) -->
  <bean
    class="org.terasoluna.batch.item.file.transform.EnclosableDelimitedLineAggregator"
      p:delimiter=","
      p:enclosure=''''
      p:allEnclosing="true">
    <property name="fieldExtractor">
      <!-- omitted settings -->
    </property>
  </bean>
</property>
```

## 設定内容の項目一覧

項目番号	プロパティ名	設定内容	必須	デフォルト値
(1)	lineAggregator	org.terasoluna.batch.item.file.transform.EnclosableDelimitedLineAggregatorを設定する。	✓	なし
(2)	delimiter	区切り文字を設定する。		カンマ
(3)	enclosure	囲み文字を設定する。 囲み文字がフィールドに含まれる場合は、エスケープ処理として囲み文字を2つ連結されたものへ置換される。		ダブルクオート
(4)	allEnclosing	trueの場合、すべてのフィールドが囲み文字で囲まれる。 falseの場合フィールド内に行頭復帰(CR)、改行(LF)、区切り文字、囲み文字が含まれるフィールドのみ囲み文字で囲まれる。		false

*EnclosableDelimitedLineAggregator*の提供について

TERASOLUNA Batch 5.xでは、RFC-4180の仕様を満たすことを目的として拡張クラス `org.terasoluna.batch.item.file.transform.EnclosableDelimitedLineAggregator` を提供している。

Spring Batchが提供してい

る `org.springframework.batch.item.file.transform.DelimitedLineAggregator` はフィールドを囲み文字で囲む処理に対応しておらず、RFC-4180の仕様を満たすことができないためである。 [Spring Batch/BATCH-2463](#) を参照のこと。



CSV形式のフォーマットについて、CSV形式の一般的書式とされるRFC-4180では下記のように定義されている。

- フィールドに改行、囲み文字、区切り文字が含まれていない場合、各フィールドはダブルクオート(囲み文字)で囲んでも囲わなくてもよい
- 改行(CRLF)、ダブルクオート(囲み文字)、カンマ(区切り文字)を含むフィールドは、ダブルクオートで囲むべきである
- フィールドがダブルクオート(囲み文字)で囲まれている場合、フィールドの値に含まれるダブルクオートは、その直前に1つダブルクオートを付加して、エスケープしなければならない

TSV形式ファイルの出力方法

TSVファイルの出力を行う場合には、区切り文字にタブを設定することで実現可能である。

TSVファイル出力時の区切り文字設定例(定数による設定)



```
<property name="delimiter">
  <util:constant
    static-
    field="org.springframework.batch.item.file.transform.DelimitedLineToke
    nizer.DELIMITER_TAB"/>
</property>
```

または、以下のようにしてもよい。

TSVファイル出力時の区切り文字設定例(文字参照による設定)

```
<property name="delimiter" value="
"/>
```

### 5.3.2.2. 固定長レコード

固定長レコードファイルを扱う場合の定義方法を説明する。

#### 5.3.2.2.1. 入力

下記の入力ファイルを読み込むための設定例を示す。

TERASOLUNA Batch 5.xでは、レコードの区切りを改行で判断する形式とバイト数で判断する形式に対応している。

#### 入力ファイル例1(レコードの区切りは改行)

```
売上012016 1 00000011000000000  
売上022017 2 00000022000000000  
売上032018 3 00000033000000000
```

#### 入力ファイル例2(レコードの区切りはバイト数、32バイトで1レコード)

```
売上012016 1 00000011000000000 売上022017 2 00000022000000000 売上032018 3  
00000033000000000
```

#### 入力ファイル仕様

項目番号	フィールド名	データ型	バイト数
(1)	branchId	String	6
(2)	year	int	4
(3)	month	int	2
(4)	customerId	String	10
(5)	amount	BigDecimal	10

#### 変換対象クラス

```
public class SalesPlanDetail {  
  
    private String branchId;  
    private int year;  
    private int month;  
    private String customerId;  
    private BigDecimal amount;  
  
    // omitted getter/setter  
}
```

上記のファイルを読む込むための設定は以下のとおり。

## Bean定義例

```
<!-- (1) (2) (3) -->
<bean id="reader"
    class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
    p:resource="#{jobParameters[inputFile]}"
    p:encoding="MS932"
    p:strict="true">
    <property name="bufferedReaderFactory"> <!-- (4) -->
        <bean class=
"org.springframework.batch.item.file.DefaultBufferedReaderFactory"/>
    </property>
    <property name="lineMapper"> <!-- (5) -->
        <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <property name="lineTokenizer"> <!-- (6) -->
                <!-- (7) -->
                <!-- (8) -->
                <!-- (9) -->
                <bean
class="org.terasoluna.batch.item.file.transform.FixedByteLengthLineTokenizer"
                p:names="branchId,year,month,customerId,amount"
                c:ranges="1-6, 7-10, 11-12, 13-22, 23-32"
                c:charset="MS932" />
            </property>
            <property name="fieldSetMapper"> <!-- (10) -->
                <bean
class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"
                p:targetType="org.terasoluna.batch.functionalttest.app.model.plan.SalesPlanDetail"/>
            </property>
        </bean>
    </property>
</bean>
```

## 設定内容の項目一覧

項目番	プロパティ名	設定内容	必須	デフォルト値
(1)	resource	入力ファイルを設定する。	✓	なし
(2)	encoding	入力ファイルの文字コードを設定する。		JavaVMのデフォルト文字セット
(3)	strict	trueを設定すると、入力ファイルが存在しない(開けない)場合に例外が発生する。		true

項目番	プロパティ名	設定内容	必須	デフォルト値
(4)	bufferedReaderFactory	<p>レコードの区切りを改行で判断する場合は、デフォルト値である<a href="#">org.springframework.batch.item.file.DefaultBufferedReaderFactory</a>を使用する。</p> <p><a href="#">DefaultBufferedReaderFactory</a>が生成する<a href="#">BufferedReader</a>は改行までを1レコードとして取得する。</p> <p>レコードの区切りをバイト数で判断する場合は、TERASOLUNA Batch 5.x が提供する<a href="#">org.terasoluna.batch.item.file.FixedByteLengthBufferedReaderFactory</a>を設定する。</p> <p><a href="#">FixedByteLengthBufferedReaderFactory</a>が生成する<a href="#">BufferedReader</a>は指定したバイト数までを1レコードとして取得する。</p> <p><a href="#">FixedByteLengthBufferedReaderFactory</a>の詳しい仕様および使用方法は後述する。</p>		DefaultBufferedReaderFactory
(5)	lineMapper	<a href="#">org.springframework.batch.item.file.mapping.DefaultLineMapper</a> を設定する。	✓	なし
(6)	lineTokenizer	TERASOLUNA Batch 5.x が提供する <a href="#">org.terasoluna.batch.item.file.transform.FixedByteLengthLineTokenizer</a> を設定する。	✓	なし
(7)	names	<p>1レコードの各項目に名前を付与する。</p> <p><a href="#">FieldSetMapper</a>で使われる<a href="#">FieldSet</a>で設定した名前を用いて各項目を取り出すことができるようになる。</p> <p>レコードの先頭から各名前をカンマ区切りで設定する。</p> <p><a href="#">BeanWrapperFieldSetMapper</a>を利用する場合は、必須設定である。</p>		なし
(8)	ranges (コンストラクタ引数)	<p>区切り位置を設定する。レコードの先頭から区切り位置をカンマ区切りで設定する。</p> <p>各区切り位置の単位はバイトであり、<a href="#">開始位置-終了位置</a>形式で指定する。</p> <p>区切り位置を設定した順番でレコードから指定された範囲を取得し、<a href="#">FieldSet</a>に格納される。</p> <p>(6)のnamesを指定した場合は区切り位置を設定した順番でnamesと対応付けて<a href="#">FieldSet</a>に格納される。</p>	✓	なし
(9)	charset (コンストラクタ引数)	(2)で指定した文字コードと同じ値を設定する。	✓	なし

項目番	プロパティ名	設定内容	必須	デフォルト値
(10)	fieldSetMapper	<p>文字列や数字など特別な変換処理が不要な場合は、<code>org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper</code>を利用し、プロパティ <code>targetType</code>に変換対象クラスを指定する。</p> <p>これにより、(6)で設定した各項目の名前と一致するフィールドに値を自動的に設定したインスタンスを生成する。</p> <p>変換処理が必要な場合は、<code>org.springframework.batch.item.file.mapping.FieldSetMapper</code>の実装クラスを設定する。</p>	✓	なし



### FieldSetMapperの独自実装について

FieldSetMapperを独自に実装する場合については、[How To Extend](#)を参照すること。

### FixedByteLengthBufferedReaderFactoryの使用方法

レコードの区切りをバイト数で判断するファイルを読み込む場合は、TERASOLUNA Batch 5.xが提供する`org.terasoluna.batch.item.file.FixedByteLengthBufferedReaderFactory`を使用する。

`FixedByteLengthBufferedReaderFactory`を使用することで指定したバイト数までを1レコードとして取得することができる。

`FixedByteLengthBufferedReaderFactory`の仕様は以下のとおり。

- コンストラクタ引数としてレコードのバイト数を指定する
- 指定されたバイト数を1レコードとしてファイルを読み込む`FixedByteLengthBufferedReader`を生成する

`FixedByteLengthBufferedReader`の使用は以下のとおり。

- インスタンス生成時に指定されたバイト長を1レコードとしてファイルを読み込む
- 改行コードが存在する場合、破棄せず1レコードのバイト長に含めて読み込みを行う
- 読み込み時に使用するファイルエンコーディングは`FlatFileItemWriter`に設定したもののが`BufferedReader`生成時に設定される

`FixedByteLengthBufferedReaderFactory`の定義方法を以下に示す。

```

<property name="bufferedReaderFactory">
    <bean class="org.terasoluna.batch.item.file.FixedByteLengthBufferedReaderFactory"
        c:byteLength="32"/> <!-- (1) -->

</property>

```

### 設定内容の項目一覧

項目番	プロパティ名	設定内容	必須	デフォルト値
(1)	byteLength (コンストラクタ引数)	1レコードあたりのバイト数を設定する。	✓	なし

固定長ファイルを扱う場合に使用するコンポーネント

固定長ファイルを扱う場合は、TERASOLUNA Batch 5.xが提供するコンポーネントを使うことを前提にしている。

*FixedByteLengthBufferedReaderFactory*



改行なし固定長ファイルから、指定した文字コードのバイト数で1レコードを読み込む*BufferedReader*生成クラス

*FixedByteLengthLineTokenizer*

マルチバイト文字列に対応したバイト数切りの*FixedLengthTokenizer*拡張クラス

マルチバイト文字列を含むレコードを処理する場合

マルチバイト文字列を含むレコードを処理する場合は、*FixedByteLengthLineTokenizer*を必ず利用する。



Spring Batchが提供する*FixedLengthTokenizer*は、レコードをバイト数ではなく文字数で区切ってしまうため、期待どおりの項目切り出しが行われない恐れがある。この点についてはJIRAの [Spring Batch/BATCH-2540](#) で報告しているため、今後不要になる可能性がある。



FieldSetMapperの実装については、[How To Extend](#)を参照すること。

### 5.3.2.2.2. 出力

下記の出力ファイルを書き出すための設定例を示す。

固定長ファイルを書き出すためには、Beanから取得した値をフィールドのバイト数にあわせてフォーマットを行う必要がある。

フォーマットの実行方法は全角文字が含まれるか否かによって下記のように異なる。

- 全角文字が含まれない場合(半角文字のみであり文字のバイト数が一定)
  - *FormatterLineAggregator*にてフォーマットを行う。
  - フォーマットは、*String.format*メソッドで使用する書式で設定する。
- 全角文字が含まれる場合(文字コードによって文字のバイト数が一定ではない)
  - *FieldExtractor*の実装クラスにてフォーマットを行う。

まず、出力ファイルに全角文字が含まれない場合の設定例を示し、その後全角文字が含まれる場合の設定例を示す。

出力ファイルに全角文字が含まれない場合の設定について下記に示す。

## 出力ファイル例

```
0012016 10000000001 10000000  
0022017 20000000002 20000000  
0032018 30000000003 30000000
```

## 出力ファイル仕様

項目番号	フィールド名	データ型	バイト数
(1)	branchId	String	6
(2)	year	int	4
(3)	month	int	2
(4)	customerId	String	10
(5)	amount	BigDecimal	10

フィールドのバイト数に満たない部分は半角スペース埋めとしている。

## 変換対象クラス

```
public class SalesPlanDetail {  
  
    private String branchId;  
    private int year;  
    private int month;  
    private String customerId;  
    private BigDecimal amount;  
  
    // omitted getter/setter  
}
```

上記のファイルを書き出すための設定は以下のとおり。

## Bean定義

```
<!-- Writer -->
<!-- (1) (2) (3) (4) (5) (6) (7) -->
<bean id="writer"
      class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step"
      p:resource="#{jobParameters[outputFile]}"
      p:encoding="MS932"
      p:lineSeparator="\n"
      p:appendAllowed="true"
      p:shouldDeleteIfEmpty="true"
      p:shouldDeleteIfExists="false"
      p:transactional="true">
    <property name="lineAggregator" > <!-- (8) -->
      <bean
        class="org.springframework.batch.item.file.transform.FormatterLineAggregator"
        p:format="%6s%4s%2s%10s%10s"/> <!-- (9) -->
        <property name="fieldExtractor" > <!-- (10) -->
          <bean
            class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"
            p:names="branchId,year,month,customerId,amount"/> <!-- (11) -->
          </property>
        </bean>
      </property>
    </bean>
  
```

## 設定内容の項目一覧

項目番号	プロパティ名	設定内容	必須	デフォルト値
(1)	resource	出力ファイルを設定する。	✓	なし
(2)	encoding	入力ファイルの文字コードを設定する。		JavaVMのデフォルト文字セット
(3)	lineSeparator	レコード区切り(改行コード)を設定する。 改行なしにする場合は、空文字を設定する。		システムプロパティのline.separator
(4)	appendAllowed	trueの場合、既存のファイルに追記をする。		false
(5)	shouldDeleteIfEmpty	trueの場合、出力結果が空ファイルであれば削除する。		false
(6)	shouldDeleteIfExists	trueの場合、既にファイルが存在すれば削除する。 falseの場合、既にファイルが存在すれば例外をスローする。		true
(7)	transactional	トランザクション制御を行うかを設定する。 詳細は、 <a href="#">トランザクション制御</a> を参照		true
(8)	lineAggregator	<a href="#">org.springframework.batch.item.file.transform.FormatterLineAggregator</a> を設定する。	✓	なし

項目番	プロパティ名	設定内容	必須	デフォルト値
(9)	format	<code>String.format</code> メソッドで使用する書式で出力フォーマットを設定する。	✓	なし
(10)	fieldExtractor	文字列や数字など特別な変換処理、全角文字のフォーマットが不要な場合は、 <code>org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor</code> が利用できる。 値の変換処理や全角文字をフォーマットする等の対応が必要な場合は、 <code>org.springframework.batch.item.file.transform.FieldExtractor</code> の実装クラスを設定する。 全角文字をフォーマットする場合における <code>FieldExtractor</code> の実装例は後述する。		<code>PassThroughFieldExtractor</code>
(11)	names	1レコードの各項目に名前を付与する。 レコードの先頭から各フィールドの名前をカンマ区切りで設定する。	✓	なし

*PassThroughFieldExtractor*とは

`FormatterLineAggregator`がもつプロパティ `fieldExtractor` のデフォルト値は `org.springframework.batch.item.file.transform.PassThroughFieldExtractor` である。



`PassThroughFieldExtractor` は、元のアイテムに対して処理を行わずに返すクラスであり、`FieldExtractor` にて何も処理を行わない場合に使用する。

アイテムが配列またはコレクションの場合はそのまま返されるが、それ以外の場合には、単一要素の配列にラップされる。

#### 全角文字が含まれるフィールドに対してフォーマットを行う際の設定例

全角文字に対するフォーマットを行う場合、文字コードにより1文字あたりのバイト数が異なるため、`FormatterLineAggregator`ではなく、`FieldExtractor`の実装クラスを使用する。

`FieldExtractor`の実装クラスは以下の要領で実装する。

- `FieldExtractor` クラスを実装し、`extract` メソッドをオーバーライドする
- `extract` メソッドは以下の要領で実装する
  - `item`(処理対象のBean)から値を取得し、適宜変換処理等を行う
  - `Object`型の配列に格納し返す

`FieldExtractor`の実装クラスで行う全角文字を含むフィールドのフォーマットは以下の要領で実装する。

- 文字コードに対するバイト数を取得する
- 取得したバイト数を元にパディング・トリム処理で整形する

以下に全角文字を含むフィールドをフォーマットする場合の設定例を示す。

## 出力ファイル例

```
0012016 10000000001 10000000  
番号2017 2 売上高002 20000000  
番号32018 3 売上003 30000000
```

出力ファイルの使用は上記の例と同様。

Bean定義(lineAggregatorの設定のみ)

```
<property name="lineAggregator"> <!-- (1) -->  
  <bean  
    class="org.springframework.batch.item.file.transform.FormatterLineAggregator"  
      p:format="%s%4s%2s%s%10s"/> <!-- (2) -->  
      <property name="fieldExtractor"> <!-- (3) -->  
        <bean  
          class="org.terasoluna.batch.functionaltest.ch05.fileaccess.plan.SalesPlanFixedLengthFi  
eldExtractor"/>  
        </property>  
      </bean>  
    </property>
```

## 設定内容の項目一覧

項目番	プロパティ名	設定内容	必須	デフォルト値
(1)	lineAggregator	org.springframework.batch.item.file.transform.FormatterLineAggregatorを設定する。	✓	なし
(2)	format	String.formatメソッドで使用する書式で出力フォーマットを設定する。 全角文字が含まれないフィールドに対してのみ桁数の指定をしている。	✓	なし
(3)	fieldExtractor	FieldExtractorの実装クラスを設定する。 実装例は後述する。		PassThroughFieldExtractor

## 変換対象クラス

```
public class SalesPlanDetail {  
  
  private String branchId;  
  private int year;  
  private int month;  
  private String customerId;  
  private BigDecimal amount;  
  
  // omitted getter/setter  
}
```

## 全角文字をフォーマットするFieldExtractorの実装例

```
public class SalesPlanFixedLengthFieldExtractor implements FieldExtractor<SalesPlanDetail> {
    // (1)
    @Override
    public Object[] extract(SalesPlanDetail item) {
        Object[] values = new Object[5]; // (2)

        // (3)
        values[0] = fillUpSpace(item.getBranchId(), 6); // (4)
        values[1] = item.getYear();
        values[2] = item.getMonth();
        values[3] = fillUpSpace(item.getCustomerId(), 10); // (4)
        values[4] = item.getAmount();

        return values; // (7)
    }

    // It is a simple impl for example
    private String fillUpSpace(String val, int num) {
        String charsetName = "MS932";
        int len;
        try {
            len = val.getBytes(charsetName).length; // (5)
        } catch (UnsupportedEncodingException e) {
            // omitted exception handling
        }

        String fillStr = "";
        for (int i = 0; i < (num - len); i++) { // (6)
            fillStr += " ";
        }

        return fillStr + val;
    }
}
```

## 設定内容の項目一覧

項目番	説明
(1)	FieldExtractorクラスを実装し、extractメソッドをオーバーライドする。 FieldExtractorの型引数には変換対象クラスを設定する。
(2)	変換処理等を行ったデータを格納するためのObject型配列を定義する。
(3)	引数で受けたitem(処理対象のBean)から値を取得し、適宜変換処理を行い、Object型の配列に格納する。
(4)	全角文字が含まれるフィールドに対してフォーマット処理を行う。 フォーマット処理の詳細は(5)、(6)を参照すること。

項目番	説明
(5)	文字コードに対するバイト数を取得する。
(6)	取得したバイト数を元にパディング・トリム処理で整形する。 実装例では指定されたバイト数まで文字列の前に空白を付与している。
(7)	処理結果を保持しているObject型の配列を返す。

### 5.3.2.3. 単一文字列レコード

单一文字列レコードファイルを扱う場合の定義方法を説明する

#### 5.3.2.3.1. 入力

下記の入力ファイルを読み込むための設定例を示す。

入力ファイル例

```
Summary1:4,000,000,000
Summary2:5,000,000,000
Summary3:6,000,000,000
```

上記のファイルを読むための設定は以下のとおり。

Bean定義

```
<!-- (1) (2) (3) -->
<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
      p:resource="#{jobParameters[inputFile]}"
      p:encoding="MS932"
      p:strict="true">
    <property name="lineMapper"> <!-- (4) -->
      <bean
        class="org.springframework.batch.item.file.mapping.PassThroughLineMapper"/>
    </property>
</bean>
```

設定内容の項目一覧

項目番	プロパティ名	設定内容	必須	デフォルト値
(1)	resource	入力ファイルを設定する。	✓	なし
(2)	encoding	入力ファイルの文字コードを設定する。		JavaVMのデフォルト文字セット
(3)	strict	trueを設定すると、入力ファイルが存在しない(開けない)場合に例外が発生する。		true

項目番	プロパティ名	設定内容	必須	デフォルト値
(4)	lineMapper	org.springframework.batch.item.file.mapping.PassThroughLineMapperを設定する。 PassThroughLineMapperは渡されたレコードをそのまま文字列として返すLineMapperの実装クラスである。	✓	なし

### 5.3.2.3.2. 出力

下記の出力ファイルを書き出すための設定例を示す。

#### 出力ファイル例

```
Summary1:4,000,000,000
Summary2:5,000,000,000
Summary3:6,000,000,000
```

#### Bean定義

```
<!-- Writer -->
<!-- (1) (2) (3) (4) (5) (6) (7) -->
<bean id="writer"
      class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step"
      p:resource="#{jobParameters[outputFile]}"
      p:encoding="MS932"
      p:lineSeparator="\n"
      p:appendAllowed="true"
      p:shouldDeleteIfEmpty="true"
      p:shouldDeleteIfExists="false"
      p:transactional="true">
    <property name="lineAggregator"> <!-- (8) -->
      <bean
        class="org.springframework.batch.item.file.transform.PassThroughLineAggregator"/>
    </property>
</bean>
```

#### 設定内容の項目一覧

項目番	プロパティ名	設定内容	必須	デフォルト値
(1)	resource	出力ファイルを設定する。	✓	なし
(2)	encoding	入力ファイルの文字コードを設定する。		JavaVMのデフォルト文字セット
(3)	lineSeparator	レコード区切り(改行コード)を設定する。		システムプロパティのline.separator
(4)	appendAllowed	trueの場合、既存のファイルに追記をする。		false

項目番	プロパティ名	設定内容	必須	デフォルト値
(5)	shouldDeleteIfEmpty	trueの場合、出力結果が空ファイルであれば削除する。		false
(6)	shouldDeleteIfExists	trueの場合、既にファイルが存在すれば削除する。 falseの場合、既にファイルが存在すれば例外をスローする。		true
(7)	transactional	トランザクション制御を行うかを設定する。 詳細は、 <a href="#">トランザクション制御</a> を参照。		true
(8)	lineAggregator	<code>org.springframework.batch.item.file.transform.PassThroughLineAggregator</code> を設定する。 PassThroughLineAggregatorはitem(処理対象のBean)をそのまま文字列へ変換( <code>item.toString()</code> を実行)するLineAggregatorの実装クラスである。	✓	なし

#### 5.3.2.4. ヘッダとフッタ

ヘッダ・フッタがある場合の入出力方法を説明する。

ここでは行数指定にてヘッダ・フッタを読み飛ばす方法を説明する。

ヘッダ・フッタのレコード数が可変であり行数指定ができない場合は、[マルチフォーマットの入力](#)を参考に[PatternMatchingCompositeLineMapper](#)を使用すること。

##### 5.3.2.4.1. 入力

ヘッダの読み飛ばし

ヘッダレコードを読み飛ばす方法には以下に示す2パターンがある。

- `FlatFileItemReader`の`linesToSkip`にファイルの先頭から読み飛ばす行数を設定
- OSコマンドによる前処理でヘッダレコードを取り除く

入力ファイル例

```
sales_plan_detail_11
branchId,year,month,customerId,amount
000001,2016,1,0000000001,1000000000
000002,2017,2,0000000002,2000000000
000003,2018,3,0000000003,3000000000
```

先頭から2行がヘッダレコードである。

上記のファイルを読む込むための設定は以下のとおり。

`linesToSkip`による読み飛ばし

```
<bean id="reader"
  class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
  p:resource="file:#{jobParameters[inputFile]}"
  p:linesToSkip=value="2" > <!-- (1) -->
<property name="lineMapper">
  <!-- omitted settings -->
</property>
</bean>
```

#### 設定内容の項目一覧

項目番	プロパティ名	設定内容	必須	デフォルト値
(1)	linesToSkip	読み飛ばすヘッダ行数を設定する。		0

#### OSコマンドによる読み飛ばし処理

```
# Remove number of lines in header from the top of input file
tail -n +'expr 2 + 1' input.txt > output.txt
```

`tail`コマンドを利用し、入力ファイル`input.txt`の3行目以降を取得し、`output.txt`に出力している。`tail`コマンドのオプション`-n +K`に指定する値はヘッダレコードの数+1となるため注意すること。

ヘッダレコードとフッタレコードを読み飛ばすOSコマンド

`head`コマンドと`tail`コマンドをうまく活用することでヘッダレコードとフッタレコードを行数指定をして読み飛ばすことが可能である。

ヘッダレコードの読み飛ばし方

`tail`コマンドをオプション`-n +K`を付与して実行することで、処理対象の`K`行目以降を取得する。

フッタレコードの読み飛ばし方

`head`コマンドをオプション`-n -K`を付与して実行することで、処理対象の末尾から`K`行目より前を取得する。

ヘッダレコードとフッタレコードをそれぞれ読み飛ばすシェルスクリプト例を下記に示す。



## ヘッダ/フッタから指定行数を取り除くシェルスクリプトの例

```
#!/bin/bash

if [ $# -ne 4 ]; then
    echo "The number of arguments must be 4, given is $#." 1>&2
    exit 1
fi

# Input file.
input=$1

# Output file.
output=$2

# Number of lines in header.
header=$3

# Number of lines in footer.
footer=$4

# Remove number of lines in header from the top of input file
# and number of lines in footer from the end,
# and save to output file.
tail -n +'expr ${header} + 1' ${input} | head -n -$footer >
${output}
```

### 引数

項目番	説明
(1)	入力ファイル
(2)	出力ファイル
(3)	読み飛ばすヘッダの行数
(4)	読み飛ばすフッタの行数

### ヘッダ情報の取り出し

ヘッダレコードを認識し、ヘッダレコードの情報を取り出す方法を示す。

ヘッダ情報の取り出しが以下の要領で実装する。

### 設定

- `org.springframework.batch.item.file.LineCallbackHandler` の実装クラスにヘッダに対する処理を実装する
  - `LineCallbackHandler#handleLine()` 内で取得したヘッダ情報を `stepExecutionContext` に格納する
  - `FlatFileItemReader` の `skippedLinesCallback` に `LineCallbackHandler` の実装クラスを設定する

- `FlatFileItemReader`の`linesToSkip`にヘッダの行数を指定する

ファイル読み込みおよびヘッダ情報の取り出し

- `linesToSkip`の設定によってスキップされるヘッダレコード1行ごとに`LineCallbackHandler#handleLine()`が呼び出される
  - ヘッダ情報が`stepExecutionContext`に格納される

取得したヘッダ情報を利用する

- ヘッダ情報を`stepExecutionContext`から取得してデータ部の処理で利用する

ヘッダレコードの情報を取り出す際の実装例を示す。

*Bean定義*

```
<bean id="lineCallbackHandler"
  class="org.terasoluna.batch.functionaltest.ch05.fileaccess.module.HoldHeaderLineCallbackHandler"/>

<!-- (1) (2) -->
<bean id="reader"
  class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
  p:linesToSkip="2"
  p:skippedLinesCallback-ref="lineCallbackHandler"
  p:resource="file:#{jobParameters[inputFile]}>
  <property name="lineMapper">
    <!-- omitted settings -->
  </property>
</bean>

<batch:job id="jobReadCsvSkipAndReferHeader" job-repository="jobRepository">
  <batch:step id="jobReadCsvSkipAndReferHeader.step01">
    <batch:tasklet transaction-manager="jobTransactionManager">
      <batch:chunk reader="reader"
        processor="loggingHeaderRecordItemProcessor"
        writer="writer" commit-interval="10"/>
      <batch:listeners>
        <batch:listener ref="lineCallbackHandler"/> <!-- (3) -->
      </batch:listeners>
    </batch:tasklet>
  </batch:step>
</batch:job>
```

設定内容の項目一覧

項目番号	プロパティ名	設定内容	必須	デフォルト値
(1)	<code>linesToSkip</code>	読み飛ばすヘッダ行数を設定する。		0

項目番	プロパティ名	設定内容	必須	デフォルト値
(2)	skippedLinesCallback	<code>LineCallbackHandler</code> の実装クラスを設定する。 実装例は後述する。		なし
(2)	listener	<code>StepExecutionListener</code> の実装クラスを設定する。 <code>FlatFileItemReader</code> の <code>skippedLinesCallback</code> に指定する <code>LineCallbackHandler</code> は自動で <code>Listener</code> として登録されないため設定が必須となる。 詳しい理由は後述する。		なし

### リスナー設定について

下記の2つの場合は自動で`Listener`として登録されないため、ジョブ定義時に`Listeners`にも定義を追加する必要がある。

(リスナーの定義を追加しないと、`StepExecutionListener#beforeStep()`が実行されない)

- `FlatFileItemReader`の`skippedLinesCallback`に指定する`LineCallbackHandler`の`StepExecutionListener`
- Taskletの実装クラスに実装する`StepExecutionListener`



```

<batch:job id="jobReadCsvSkipAndReferHeader" job-
repository="jobRepository">
    <batch:step id="jobReadCsvSkipAndReferHeader.step01">
        <batch:tasklet transaction-manager=
"jobTransactionManager">
            <batch:chunk reader="reader"

processor="loggingHeaderRecordItemProcessor"
                writer="writer" commit-interval="10"/>
        <batch:listeners>
            <batch:listener ref="loggingItemReaderListener"/>
            <!-- mandatory -->
            <batch:listener ref="lineCallbackHandler"/>
        </batch:listeners>
    </batch:tasklet>
</batch:step>
</batch:job>

```

`LineCallbackHandler`は以下の要領で実装する。

- `StepExecutionListener#beforeStep()`の実装
  - 下記のいずれかの方法で`StepExecutionListener#beforeStep()`を実装する
    - `StepExecutionListener`クラスを実装し、`beforeStep`メソッドをオーバーライドする
    - `beforeStep`メソッドを実装し、`@BeforeStep`アノテーションを付与する

- beforeStepメソッドにてStepExecutionを取得してクラスフィールドに保持する
- LineCallbackHandler#handleLine()の実装
  - LineCallbackHandlerクラスを実装し、handleLineメソッドをオーバーライドする
    - handleLineメソッドはスキップする1行ごとに1回呼ばれる点に注意すること。
  - StepExecutionからstepExecutionContextを取得し、stepExecutionContextにヘッダ情報を格納する。

### LineCallbackHandlerの実装例

```
@Component
public class HoldHeaderLineCallbackHandler implements LineCallbackHandler { // (!)
    private StepExecution stepExecution; // (2)

    @BeforeStep // (3)
    public void beforeStep(StepExecution stepExecution) {
        this.stepExecution = stepExecution; // (4)
    }

    @Override // (5)
    public void handleLine(String line) {
        this.stepExecution.getExecutionContext().putString("header", line); // (6)
    }
}
```

### 設定内容の項目一覧

項目番	説明
(1)	LineCallbackHandlerクラスを実装し、handleLineメソッドをオーバーライドする。
(2)	StepExecutionを保持するためのフィールドを定義する。
(3)	beforeStepメソッドを実装し、@BeforeStepアノテーションを付与する。 シグネチャはvoid beforeStep(StepExecution stepExecution)とする。 StepExecutionListenerクラスを実装し、beforeStepメソッドをオーバーライドする方法でもよい。
(4)	StepExecutionを取得してクラスフィールドに保持する。
(5)	LineCallbackHandlerクラスを実装し、handleLineメソッドをオーバーライドする。
(6)	StepExecutionからstepExecutionContextを取得し、headerというキーを指定してstepExecutionContextにヘッダ情報を格納する。 ここでは簡単のため、スキップする2行のうち、最後の1行だけを格納している。

ヘッダ情報をstepExecutionContextから取得してデータ部の処理で利用する例を示す。

ItemProcessorにてヘッダ情報を利用する場合を例にあげて説明する。

他のコンポーネントでヘッダ情報を利用する際も同じ要領で実現することができる。

ヘッダ情報を利用する処理は以下の要領で実装する。

- `LineCallbackHandler`の実装例と同様に`StepExecutionListener#beforeStep()`を実装する
- `beforeStep`メソッドにて`StepExecution`を取得してクラスフィールドに保持する
- `StepExecution`から`stepExecutionContext`およびヘッダ情報を取得して利用する

ヘッダ情報の利用例

```
@Component
public class LoggingHeaderRecordItemProcessor implements
    ItemProcessor<SalesPlanDetail, SalesPlanDetail> {
    private StepExecution stepExecution; // (1)

    @BeforeStep // (2)
    public void beforeStep(StepExecution stepExecution) {
        this.stepExecution = stepExecution; // (3)
    }

    @Override
    public SalesPlanDetail process(SalesPlanDetail item) throws Exception {
        String headerData = this.stepExecution.getExecutionContext()
            .getString("header"); // (4)
        // omitted business logic
        return item;
    }
}
```

設定内容の項目一覧

項目番	説明
(1)	<code>StepExecution</code> を保持するためのフィールドを定義する。
(2)	<code>beforeStep</code> メソッドを実装し、 <code>@BeforeStep</code> アノテーションを付与する。 シグネチャは <code>void beforeStep(StepExecution stepExecution)</code> とする。 <code>StepExecutionListener</code> クラスを実装し、 <code>beforeStep</code> メソッドをオーバーライドする方法でもよい。
(3)	<code>StepExecution</code> を取得してクラスフィールドに保持する。
(4)	<code>StepExecution</code> から <code>stepExecutionContext</code> を取得し、 <code>header</code> というキーを指定して <code>stepExecutionContext</code> からヘッダ情報を取得する。

### Job/StepのExecutionContextの使用について

ヘッダ(フッタ)情報の取り出しへは、読み込んだヘッダ情報をStepExecutionのExecutionContextに格納しておき、使用する際にExecutionContextから取り出す方をとる。



下記の例では1つのステップ内でヘッダ情報の取得および利用を行うためStepExecutionのExecutionContextへヘッダ情報を格納している。ヘッダ情報の取得および利用にてステップが分かれる場合はJobExecutionのExecutionContextを利用すること。

Job/StepのExecutionContextに関する詳細は、[Spring Batchのアーキテクチャ](#)を参照すること。

### フッタの呼び飛ばし

Spring BatchおよびTERASOLUNA Batch 5.xでは、フッタレコードの読み飛ばし機能は提供していないため、OSコマンドで対応する。

#### 入力ファイル例

```
000001,2016,1,0000000001,1000000000  
000002,2017,2,0000000002,2000000000  
000003,2018,3,0000000003,3000000000  
number of items,3  
total of amounts,6000000000
```

末尾から2行がフッタレコードである。

上記のファイルを読む込むための設定は以下のとおり。

#### OSコマンドによる読み飛ばし処理

```
# Remove number of lines in footer from the end of input file  
head -n -2 input.txt > output.txt
```

headコマンドを利用し、入力ファイルinput.txtの末尾から2行目より前を取得し、output.txtに出力している。



フッタレコードを読み飛ばすことは、Spring Batchでは機能をもっていないため、JIRAの[Spring Batch/BATCH-2539](#)にて報告している。

よって、今後OSコマンドによる読み飛ばしだけではなく、Spring Batchでも対応できるようになる可能性がある。

### フッタ情報の取り出し

Spring BatchおよびTERASOLUNA Batch 5.xでは、フッタレコードの読み飛ばし機能、フッタ情報の取得機能は提供していない。

そのため、処理を下記のようにOSコマンドによる前処理と2つのステップのに分割することで対応する。

- OSコマンドによってフッタレコードを分割する
- 1つめのステップにてフッタレコードを読み込み、フッタ情報をExecutionContextに格納する
- 2つめのステップにてExecutionContextからフッタ情報を取得し、利用する

フッタ情報を取り出しあは以下の要領で実装する。

OSコマンドによるフッタレコードの分割

- OSコマンドを利用して入力ファイルをフッタ部とフッタ部以外に分割する

1つめのステップでフッタレコードを読み込み、フッタ情報を取得する

- フッタレコードを読み込みjobExecutionContextに格納する
  - フッタ情報の格納と利用にてステップが異なるため、jobExecutionContextに格納する。
  - jobExecutionContextを利用する方法は、JobとStepのスコープに関する違い以外は、[ヘッダ情報の取り出し](#)にて説明したstepExecutionContextと同様である。

2つめのステップにて取得したフッタ情報を利用する

- フッタ情報をjobExecutionContextから取得してデータ部の処理で利用する

以下に示すファイルのフッタ情報を取り出して利用する場合を例にあげて説明する。

入力ファイル例

```
000001,2016,1,0000000001,1000000000
000002,2017,2,0000000002,2000000000
000003,2018,3,0000000003,3000000000
number of items,3
total of amounts,6000000000
```

末尾から2行がフッタレコードである。

OSコマンドによるフッタレコードの分割

上記のファイルをOSコマンドを利用してフッタ部とフッタ部以外に分割する設定は以下のとおり。

OSコマンドによる読み飛ばし処理

```
# Extract non-footer record from input file and save to output file.
head -n -2 input.txt > input_data.txt

# Extract footer record from input file and save to output file.
tail -n 2 input.txt > input_footer.txt
```

headコマンドを利用し、入力ファイルinput.txtのフッタ部以外をinput\_data.txtへ、フッタ部をinput\_footer.txtに出力している。

出力ファイル例は以下のとおり。

### 出力ファイル例(*input\_data.txt*)

```
000001,2016,1,0000000001,1000000000  
000002,2017,2,0000000002,2000000000  
000003,2018,3,0000000003,3000000000
```

### 出力ファイル例(*input\_footer.txt*)

```
number of items,3  
total of amounts,6000000000
```

### フッタ情報の取得、利用

OSコマンドにて分割したフッタレコードからフッタ情報を取得、利用する方法を説明する。

フッタレコードを読み込むステップを前処理として主処理とステップを分割している。  
ステップの分割に関する詳細は、[フロー制御](#)を参照すること。

下記の例ではフッタ情報を取得し、[jobExecutionContext](#)へフッタ情報を格納するまでの例を示す。

[jobExecutionContext](#)からフッタ情報を取得し利用する方法は[ヘッダ情報の取り出し](#)と同じ要領で実現可能である。

### データレコードの情報を保持するクラス

```
public class SalesPlanDetail {  
  
    private String branchId;  
    private int year;  
    private int month;  
    private String customerId;  
    private BigDecimal amount;  
  
    // omitted getter/setter  
}
```

### フッタレコードの情報を保持するクラス

```
public class SalesPlanDetailFooter implements Serializable {  
  
    // omitted serialVersionUID  
  
    private String name;  
    private String value;  
  
    // omitted getter/setter  
}
```

下記の要領でBean定義を行う。

- ・ フッタレコードを読み込むItemReaderを定義する
- ・ データレコードを読み込むItemReaderを定義する
- ・ フッタレコードを取得するビジネスロジックを定義する
  - ・ 下記の例ではTaskletの実装クラスで実現している
- ・ ジョブを定義する
  - ・ フッタ情報を取得する前処理ステップとデータレコードを読み込み後処理を行うステップを定義する

## Bean定義

```
<!-- ItemReader for reading footer records -->
<!-- (1) -->
<bean id="footerReader"
    class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
    p:resource="file:#{jobParameters[footerInputFile]}>
    <property name="lineMapper">
        <!-- omitted other settings -->
    </property>
</bean>

<!-- ItemReader for reading data records -->
<!-- (2) -->
<bean id="dataReader"
    class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
    p:resource="file:#{jobParameters[dataInputFile]}>
    <property name="lineMapper">
        <!-- omitted other settings -->
    </property>
</bean>

<bean id="writer"
    class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step">
    <!-- omitted settings -->
</bean>

<!-- Tasklet for reading footer records -->
<bean id="readFooterTasklet"
    class="org.terasoluna.batch.functionaltest.ch05.fileaccess.module.ReadFooterTasklet"/>

<batch:job id="jobReadAndWriteCsvWithFooter" job-repository="jobRepository">
    <!-- (3) -->
    <batch:step id="jobReadAndWriteCsvWithFooter.step01"
        next="jobReadAndWriteCsvWithFooter.step02">
        <batch:tasklet ref="readFooterTasklet"
            transaction-manager="jobTransactionManager"/>
    </batch:step>
    <!-- (4) -->
    <batch:step id="jobReadAndWriteCsvWithFooter.step02">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="dataReader"
                writer="writer" commit-interval="10"/>
        </batch:tasklet>
    </batch:step>
    <batch:listeners>
        <batch:listener ref="readFooterTasklet"/> <!-- (5) -->
    </batch:listeners>
</batch:job>
```

## 設定内容の項目一覧

項目番	項目	設定内容	必須	デフォルト値
(1)	footerReader	フッタレコードを保持するファイルを読み込むためのItemReaderを定義する。 フッタ情報を取得するステップで実行されるreadFooterTaskletにてインジェクトして使用する。		
(2)	dataReader	データレコードを保持するファイルを読み込むためのItemReaderを定義する。		
(3)	前処理ステップ	フッタ情報を取得するステップを定義する。 処理はreadFooterTaskletに実装している。実装例は後述する。		
(4)	主処理ステップ	データ情報を取得するとともにフッタ情報を利用するステップを定義する。 readerにはdataReaderを使用する。 例ではフッタ情報をjobExecutionContextから取得し利用する処理(ItemProcessor等)は実装していない。 フッタ情報を取得し利用する方法はヘッダ情報の取り出しと同じ要領で実現可能である。		
(5)	listeners	readFooterTaskletを設定する。 この設定を行わないとreadFooterTasklet内に実装するJobExecutionListener#beforeJob()が実行されない。 詳しい理由は、ヘッダ情報の取り出しを参照すること。		なし

フッタレコードを保持するファイルを読み込み、jobExecutionContextに格納する処理を行う処理の例を示す。

Taskletの実装クラスとして実現する際の要領は以下のとおり。

- Bean定義したfooterReaderを@Injectアノテーションと@Namedアノテーションを使用し名前指定でインジェクトする。
- 読み込んだフッタ情報をjobExecutionContextに格納する
  - 実現方法はヘッダ情報の取り出しと同様である

## フッタ情報の取得

```
public class ReadFooterTasklet implements Tasklet {  
    // (1)  
    @Inject  
    @Named("footerReader")  
    ItemStreamReader<SalesPlanDetailFooter> itemReader;  
  
    private JobExecution jobExecution;  
  
    @BeforeJob  
    public void beforeJob(JobExecution jobExecution) {  
        this.jobExecution = jobExecution;  
    }  
  
    @Override  
    public RepeatStatus execute(StepContribution contribution,  
                               ChunkContext chunkContext) throws Exception {  
        ArrayList<SalesPlanDetailFooter> footers = new ArrayList<>();  
  
        // (2)  
        itemReader.open(chunkContext.getStepContext().getStepExecution()  
                      .getExecutionContext());  
  
        SalesPlanDetailFooter footer;  
        while ((footer = itemReader.read()) != null) {  
            footers.add(footer);  
        }  
  
        // (3)  
        jobExecution.getExecutionContext().put("footers", footers);  
  
        return RepeatStatus.FINISHED;  
    }  
}
```

## 設定内容の項目一覧

項目番	説明
(1)	Bean定義したfooterReaderを@Injectアノテーションと@Namedアノテーションを使用し名前指定でインジェクトする。
(2)	footerReaderを使用してフッタレコードを保持したファイルを読み込みフッタ情報を取得する。 Taskletの実装クラス内でBean定義したItemReaderを使用する方法は <a href="#">タスクレット指向ジョブの作成</a> を参照すること。
(3)	JobExecutionからjobExecutionContextを取得し、footersというキーを指定してjobExecutionContextへフッタ情報を格納する。

#### 5.3.2.4.2. 出力

##### ヘッダ情報の出力

フラットファイルでヘッダ情報を出力する際は以下の要領で実装する。

- `org.springframework.batch.item.file.FlatFileHeaderCallback`の実装を行う
- 実装した`FlatFileHeaderCallback`を`FlatFileItemWriter`の`headerCallback`に設定する
  - `headerCallback`を設定すると`FlatFileItemWriter`の出力処理で、最初に`FlatFileHeaderCallback#writeHeader()`が実行される

`FlatFileHeaderCallback`は以下の要領で実装する。

- `FlatFileHeaderCallback`クラスを実装し、`writeFooter`メソッドをオーバーライドする
- 引数で受ける`Writer`を用いてフッタ情報を出力する。

下記に`FlatFileHeaderCallback`クラスの実装例を示す。

*FlatFileHeaderCallback*の実装例

```
@Component
// (1)
public class WriteHeaderFlatFileFooterCallback implements FlatFileHeaderCallback {
    @Override
    public void writeHeader(Writer writer) throws IOException {
        // (2)
        writer.write("omitted");
    }
}
```

##### 設定内容の項目一覧

項目番	説明
(1)	<code>FlatFileHeaderCallback</code> クラスを実装し、 <code>writeHeader</code> メソッドをオーバーライドする。
(2)	引数で受ける <code>Writer</code> を用いてフッタ情報を出力する。 <code>FlatFileHeaderCallback#writeHeader()</code> の実行直後に <code>FlatFileItemWriter</code> が出力する処理を実行する。 そのため、ヘッダ情報末尾の改行は出力不要である。出力される改行は、Bean定義時に <code>FlatFileItemWriter</code> に指定したものである。

## Bean定義

```
<!-- (1) (2) -->
<bean id="writer"
    class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step"
    p:headerCallback-ref="writeHeaderFlatFileFooterCallback"
    p:lineSeparator="\n"
    p:resource="file:#{jobParameters[outputFile]}>
<property name="lineAggregator">
    <!-- omitted settings -->
</property>
</bean>
```

### 設定内容の項目一覧

項目番	プロパティ名	設定内容	必須	デフォルト値
(1)	headerCallback	FlatFileHeaderCallbackの実装クラスを設定する。		
(2)	lineSeparator	レコード区切り(改行コード)を設定する。		システムプロパティのline.separator

FlatFileHeaderCallback実装時にヘッダ情報末尾の改行は出力不要



FlatFileItemWriter内でFlatFileHeaderCallback#writeHeader()の実行直後にBean定義時に指定した改行を出力する処理が実行されるため、ヘッダ情報末尾の改行は出力不要である。

### フッタ情報の出力

フラットファイルでフッタ情報を出力する際は以下の要領で実装する。

- org.springframework.batch.item.file.FlatFileFooterCallbackの実装を行う
- 実装したFlatFileFooterCallbackをFlatFileItemWriterのfooterCallbackに設定する
  - footerCallbackを設定するとFlatFileItemWriterの出力処理で、最後にFlatFileFooterCallback#writeFooter()が実行される

フラットファイルでフッタ情報を出力する方法について説明する。

FlatFileFooterCallbackは以下の要領で実装する。

- 引数で受けるWriterを用いてフッタ情報を出力する。
- FlatFileFooterCallbackクラスを実装し、writeFooterメソッドをオーバーライドする

下記にJobのExecutionContextからフッタ情報を取得し、ファイルへ出力するFlatFileFooterCallbackクラスの実装例を示す。

フッタレコードの情報を保持するクラス

```
public class SalesPlanDetailFooter implements Serializable {  
  
    // omitted serialVersionUID  
  
    private String name;  
    private String value;  
  
    // omitted getter/setter  
}
```

FlatFileFooterCallbackの実装例

```
@Component  
public class WriteFooterFlatFileFooterCallback implements FlatFileFooterCallback { // (1)  
    private JobExecution jobExecution;  
  
    @BeforeJob  
    public void beforeJob(JobExecution jobExecution) {  
        this.jobExecution = jobExecution;  
    }  
  
    @Override  
    public void writeFooter(Writer writer) throws IOException {  
        @SuppressWarnings("unchecked")  
        ArrayList<SalesPlanDetailFooter> footers = (ArrayList<SalesPlanDetailFooter>)  
this.jobExecution.getExecutionContext().get("footers"); // (2)  
  
        BufferedWriter bufferedWriter = new BufferedWriter(writer); // (3)  
        // (4)  
        for (SalesPlanDetailFooter footer : footers) {  
            bufferedWriter.write(footer.getName() + " is " + footer.getValue());  
            bufferedWriter.newLine();  
            bufferedWriter.flush();  
        }  
    }  
}
```

設定内容の項目一覧

項目番号	説明
(1)	FlatFileFooterCallbackクラスを実装し、writeFooterメソッドをオーバーライドする。
(2)	JobのExecutionContextからfootersというkeyを指定してフッタ情報を取得する。 例ではArrayListで複数のフッタ情報を取得している。
(3)	例では改行の出力にBufferedWriter.newLine()を使用するため、引数で受けるWriterを引数としてBufferedWriterを生成する。

項目番	説明
(4)	引数で受けるWriterを用いてフッタ情報を出力する。

### Bean定義

```
<bean id="writer"
      class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step"
      p:resource="#{jobParameters[outputFile]}"
      p:footerCallback-ref="writeFooterFlatFileFooterCallback" <!-- (1) -->
      <property name="lineAggregator">
          <!-- omitted settings -->
      </property>
  </bean>
```

### 設定内容の項目一覧

項目番	プロパティ名	設定内容	必須	デフォルト値
(1)	footerCallback	FlatFileFooterCallbackの実装クラスを設定する。		

### 5.3.2.5. 複数ファイル

複数ファイルを扱う場合の定義方法を説明する。

#### 5.3.2.5.1. 入力

同一レコード形式の複数ファイルを読み込む場合

は、org.springframework.batch.item.file.MultiResourceItemReaderを利用する。

MultiResourceItemReaderは指定されたItemReaderを使用し正規表現で指定された複数のファイルを読み込むことができる。

MultiResourceItemReaderは以下の要領で定義する。

- MultiResourceItemReaderのBeanを定義する
  - プロパティresourcesに読み込み対象のファイルを指定する
    - 正規表現で複数ファイルを指定する
  - プロパティdelegateにファイル読み込みに利用するItemReaderを指定する

下記に示す複数のファイルを読み込むMultiResourceItemReaderの定義例は以下のとおりである。

読み込み対象ファイル(ファイル名)

```
sales_plan_detail_01.csv
sales_plan_detail_02.csv
sales_plan_detail_03.csv
```

## Bean定義

```
<!-- (1) (2) -->
<bean id="multiResourceReader"
    class="org.springframework.batch.item.file.MultiResourceItemReader"
    scope="step"
    p:resources="file:input/sales_plan_detail_*.csv"
    p:delegate-ref="reader"/>
</bean>

<!-- (3) -->
<bean id="reader"
    class="org.springframework.batch.item.file.FlatFileItemReader">
    <property name="lineMapper">
        <!-- omitted settings -->
    </property>
</bean>
```

### 設定内容の項目一覧

項目番	プロパティ名	設定内容	必須	デフォルト値
(1)	resource	正規表現で複数の入力ファイルを設定する。	✓	なし
(2)	delegate	実際にファイルを読み込み処理するItemReaderを設定する。	✓	なし
(3)	実際にファイルを読み込み処理するItemReader	プロパティresourceは、MultiResourceItemReaderから自動的に設定されるためBean定義に設定は不要である。	✓	

MultiResourceItemReaderが使用するItemReaderにresourceの指定は不要である



MultiResourceItemReaderから委譲されるItemReaderのresourceは、MultiResourceItemReaderから自動的に設定されるためBean定義に設定は不要である。

### 5.3.2.5.2. 出力

複数ファイルを扱う場合の定義方法を説明する。

一定の件数ごとに異なるファイルへ出力する場合

は、org.springframework.batch.item.file.MultiResourceItemWriterを利用する。

MultiResourceItemWriterは指定されたItemWriterを使用して指定した件数ごとに複数ファイルへ出力することができる。

出力対象のファイル名は重複しないように一意にする必要があるが、そのための仕組みとしてResourceSuffixCreatorが提供されている。

ResourceSuffixCreatorはファイル名が一意となるようなサフィックスを生成するクラスである。

たとえば、出力対象ファイルをoutputDir/customer\_list\_01.csv(01の部分は連番)というファイル名にしたい場合は下記のように設定する。

- `MultiResourceItemWriter`に`outputDir/customer_list_`と設定する
- サフィックス`01.csv`(`01`の部分は連番)を生成する処理を`ResourceSuffixCreator`に実装する
  - 連番は`MultiResourceItemWriter`から自動で増分されて渡される値を使用することができる
- 実際に使用される`ItemWriter`には`outputDir/customer_list_01.csv`が設定される

`MultiResourceItemWriter`は以下の要領で定義する。`ResourceSuffixCreator`の実装方法は後述する。

- `ResourceSuffixCreator`の実装クラスを定義する
- `MultiResourceItemWriter`のBeanを定義する
  - プロパティ`resources`に出力対象のファイルを指定する
    - `ResourceSuffixCreator`の実装クラスで付与するサフィックスまでを設定
  - プロパティ`resourceSuffixCreator-ref`にサフィックスを生成する`ResourceSuffixCreator`の実装クラスを指定する
  - プロパティ`delegate-ref`にファイル読み込みに利用する`ItemWriter`を指定する
  - プロパティ`itemCountLimitPerResource`に1ファイルあたりの出力件数を指定する

### Bean定義

```
<!-- (1) (2) (3) (4) -->
<bean id="multiResourceItemWriter"
      class="org.springframework.batch.item.file.MultiResourceItemWriter"
      scope="step"
      p:resource="file:#{jobParameters[outputDir]}"
      p:resourceSuffixCreator-ref="customerListResourceSuffixCreator"
      p:delegate-ref="writer"
      p:itemCountLimitPerResource="4"/>
</bean>

<!-- (5) -->
<bean id="writer"
      class="org.springframework.batch.item.file.FlatFileItemWriter">
    <property name="lineAggregator">
      <!-- omitted settings -->
    </property>
</bean>

<bean id="customerListResourceSuffixCreator"
      class="org.terasoluna.batch.functionaltest.ch05.fileaccess.module.CustomerListResource
      SuffixCreator"/> <!-- (6) -->
```

### 設定内容の項目一覧

項目番	プロパティ名	設定内容	必須	デフォルト値
(1)	resource	出力対象ファイルのサフィックスを付与する前の状態を設定する。 <i>ItemWriter</i> には、 <i>MultiResourceItemWriter</i> が自動でサフィックスを付与したものが設定される。	✓	なし
(2)	resourceSuffixCreator	<i>ResourceSuffixCreator</i> の実装クラスを設定する。 デフォルト値は". " + <i>index</i> というサフィックスを生成する <i>org.springframework.batch.item.file.SimpleResourceSuffixCreator</i> である。		SimpleResourceSuffixCreator
(3)	delegate	実際にファイルを読み込み処理する <i>ItemWriter</i> を設定する。	✓	なし
(4)	itemCountLimitPerResource	1ファイルあたりの出力件数を設定する。		<i>Integer.MAX_VALUE</i>
(5)	<i>ResourceSuffixCreator</i> の実装クラス	サフィックスを生成する <i>ResourceSuffixCreator</i> の実装クラスを定義する。 実装方法は後述する。		
(6)	実際にファイルを読み込み処理する <i>ItemWriter</i>	プロパティ <i>resource</i> は、 <i>MultiResourceItemWriter</i> から自動的に設定されるためBean定義に設定は不要である。	✓	

*MultiResourceItemWriter*が使用する*ItemWriter*に *resource* の指定は不要である



*MultiResourceItemWriter*から委譲される *ItemWriter* の *resource* は、*MultiResourceItemWriter* から自動的に設定されるため Bean 定義に設定は不要である。

*ResourceSuffixCreator* は以下の要領で実装する。

- *ResourceSuffixCreator* クラスを実装し、*getSuffix* メソッドをオーバーライドする
- 引数で受ける *index* を用いてサフィックスを生成して返り値として返す
  - *index* は初期値 1 で始まり出力対象ファイルごとにインデントされる *int* 型の値である

*ResourceSuffixCreator* の実装例

```
// (1)
public class CustomerListResourceSuffixCreator implements ResourceSuffixCreator {
    @Override
    public String getSuffix(int index) {
        return String.format("%02d", index) + ".csv"; // (2)
    }
}
```

設定内容の項目一覧

項目番	説明
(1)	ResourceSuffixCreatorクラスを実装し、getSuffixメソッドをオーバーライドする。
(2)	引数で受けるindexを用いてサフィックスを生成して返り値として返す。indexは初期値1で始まり出力対象ファイルごとにインデントされるint型の値である。

### 5.3.2.6. コントロールブレイク

コントロールブレイクの実現方法について説明する。

コントロールブレイクとは

コントロールブレイク処理(またはキープレイク処理)とは、ソート済みのレコードを順次読み込み、レコード内にある特定の項目(キー項目)が同じレコードを1つのグループとして処理することを指す。

主にデータを集計するときに用いられ、キー項目が同じ値の間は集計を続け、キー項目が異なる値になる際に集計値を出力する、というアルゴリズムになる。

コントロールブレイク処理をするためには、グループの変わり目を判定するために、レコードを先読みする必要がある。`org.springframework.batch.item.support.SingleItemPeekableItemReader`を使うことで先読みを実現できる。

また、コントロールブレイクはタスクレットモデルでのみ処理可能とする。これは、チャンクが前提とする「1行で定義するデータ構造をN行処理する」や「一定件数ごとのトランザクション境界」といった点が、コントロールブレイクの「グループの変わり目で処理をする」という点と合わないためである。

コントロールブレイク処理の実行タイミングと比較条件を以下に示す。

- 対象レコード処理前にコントロールブレイク実施
  - 前回読み取ったレコードを保持し、前回レコードと現在読み込んだレコードとの比較
- 対象レコード処理後にコントロールブレイク実施
  - `SingleItemPeekableItemReader`により次のレコードを先読みし、次レコードと現在読み込んだレコードとの比較

下記にの入力データから処理結果を出力するコントロールブレイクの実装例を示す。

入力データ

```
01,2016,10,1000
01,2016,11,1500
01,2016,12,1300
02,2016,12,900
02,2016,12,1200
```

## 処理結果

```
Header Branch Id : 01,,,  
01,2016,10,1000  
01,2016,11,1500  
01,2016,12,1300  
Summary Branch Id : 01,,,3800  
Header Branch Id : 02,,,  
02,2016,12,900  
02,2016,12,1200  
Summary Branch Id : 02,,,2100
```

## コントロールブレイクの実装例

```
@Component  
public class ControlBreakTasklet implements Tasklet {  
  
    @Inject  
    SingleItemPeekableItemReader<SalesPerformanceDetail> reader; // (1)  
  
    @Inject  
    ItemStreamWriter<SalesPerformanceDetail> writer;  
  
    @Override  
    public RepeatStatus execute(StepContribution contribution,  
                               ChunkContext chunkContext) throws Exception {  
  
        // omitted.  
  
        SalesPerformanceDetail previousData = null; // (2)  
        BigDecimal summary = new BigDecimal(0); // (3)  
  
        List<SalesPerformanceDetail> items = new ArrayList<>(); // (4)  
  
        try {  
            reader.open(executionContext);  
            writer.open(executionContext);  
  
            while (reader.peek() != null) { // (5)  
                SalesPerformanceDetail data = reader.read(); // (6)  
  
                // (7)  
                if (isBreakByBranchId(previousData, data)) {  
                    SalesPerformanceDetail beforeBreakData =  
                        new SalesPerformanceDetail();  
                    beforeBreakData.setBranchId("Header Branch Id : "  
                        + currentData.getBranchId());  
                    items.add(beforeBreakData);  
                }  
            }  
        }  
    }  
}
```

```

        // omitted.
        items.add(data); // (8)

        SalesPerformanceDetail nextData = reader.peek(); // (9)
        summary = summary.add(data.getAmount());

        // (10)
        SalesPerformanceDetail afterBreakData = null;
        if (isBreakByBranchId(nextData, data)) {
            afterBreakData = new SalesPerformanceDetail();
            afterBreakData.setBranchId("Summary Branch Id : "
                + currentData.getBranchId());
            afterBreakData.setAmount(summary);
            items.add(afterBreakData);
            summary = new BigDecimal(0);
            writer.write(items); // (11)
            items.clear();
        }
        previousData = data; // (12)
    }
} finally {
    try {
        reader.close();
    } catch (ItemStreamException e) {
    }
    try {
        writer.close();
    } catch (ItemStreamException e) {
    }
}
return RepeatStatus.FINISHED;
}
// (13)
private boolean isBreakByBranchId(SalesPerformanceDetail o1,
    SalesPerformanceDetail o2) {
    return (o1 == null || !o1.getBranchId().equals(o2.getBranchId()));
}
}

```

## 設定内容の項目一覧

項目番	説明
(1)	SingleItemPeekableItemReaderをInjectする。
(2)	前回読み取ったレコードを保持する変数を定義する。
(3)	グループごとの集計値を格納する変数を定義する。
(4)	コントロールブレイクの処理結果を含めたグループ単位のレコードを格納する変数を定義する。
(5)	入力データが無くなるまで処理を繰り返す。

項目番	説明
(6)	処理対象のレコードを読み込む。
(7)	対象レコード処理前にコントロールブレイクを実施する。 ここではグループの先頭であれば見出しを設定して、(4)で定義した変数に格納する。
(8)	対象レコードへの処理結果を(4)で定義した変数に格納する。
(9)	次のレコードを先読みする。
(10)	対象レコード処理後にコントロールブレイクを実施する。 ここではグループの末尾であれば集計データをトレーラに設定して、(4)で定義した変数に格納する。
(11)	グループ単位で処理結果を出力する。
(12)	処理レコードを(2)で定義した変数に格納する。
(13)	キー項目が切り替わったか判定する。

### Bean定義

```

<!-- (1) -->
<bean id="reader"
      class="org.springframework.batch.item.support.SingleItemPeekableItemReader"
      p:delegate-ref="delegateReader" /> <!-- (2) -->

<!-- (3) -->
<bean id="delegateReader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
      p:resource="file:#{jobParameters[inputFile]}>
      <property name="lineMapper">
          <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
              <property name="lineTokenizer">
                  <bean
class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
                      p:names="branchId,year,month,customerId,amount"/>
              </property>
              <property name="fieldSetMapper">
                  <bean
class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"
p:targetType="org.terasoluna.batch.functionaltest.app.model.performance.SalesPerformanceDetail"/>
              </property>
          </bean>
      </property>
  </bean>
</bean>

```

### 設定内容の項目一覧

項目番	説明
(1)	<code>SingleItemPeekableItemReader</code> をBean定義する。TaskletへのInject対象。
(2)	<code>delegate</code> プロパティに実際にファイルを読み込むItemReaderのBeanを指定する。
(3)	実際にファイルを読み込むItemReaderのBeanを定義する。

### 5.3.3. How To Extend

ここでは、以下のケースについて説明する。

- `FieldSetMapper`の実装
- XMLファイルの入出力
- マルチフォーマットの入出力

#### 5.3.3.1. `FieldSetMapper`の実装

`FieldSetMapper`を自作で実装する方法について説明する。

`FieldSetMapper`の実装クラスは下記の要領で実装する。

- `FieldSetMapper`クラスを実装し、`mapFieldSet`メソッドをオーバーライドする
- 引数で受けた`FieldSet`から値を取得し、適宜変換処理を行い、変換対象のBeanに格納し返り値として返す
  - `FieldSet`クラスはJDBCにある`ResultSet`クラスのようにインデックスまたは名前と関連付けてデータを保持するクラスである
  - `FieldSet`クラスは`LineTokenizer`によって分割されたレコードの各フィールドの値を保持する
  - インデックスまたは名前を指定して値を格納および取得することができる

下記のような和暦フォーマットのDate型やカンマを含むBigDecimal型の変換を行うファイルを読み込む場合の実装例を示す。

入力ファイル例

```
"000001", "平成28年1月1日", "000000001", "1,000,000,000"
"000002", "平成29年2月2日", "000000002", "2,000,000,000"
"000003", "平成30年3月3日", "000000003", "3,000,000,000"
```

入力ファイル仕様

項目番	フィールド名	データ型	備考
(1)	branchId	String	
(2)	日付	Date	和暦フォーマット
(3)	customerId	String	
(4)	amount	BigDecimal	カンマを含む

## 変換対象クラス

```
public class UseDateSalesPlanDetail {  
  
    private String branchId;  
    private Date date;  
    private String customerId;  
    private BigDecimal amount;  
  
    // omitted getter/setter  
}
```

## FieldSetMapperの実装例

```
@Component
public class UseDateSalesPlanDetailFieldSetMapper implements FieldSetMapper<UseDateSalesPlanDetail> { // (1)
    /**
     * {@inheritDoc}
     *
     * @param fieldSet {@inheritDoc}
     * @return Sales performance detail.
     * @throws BindException {@inheritDoc}
     */
    @Override
    public UseDateSalesPlanDetail mapFieldSet(FieldSet fieldSet) throws BindException
    {
        UseDateSalesPlanDetail item = new UseDateSalesPlanDetail(); // (2)

        item.setBranchId(fieldSet.readString("branchId")); // (3)

        // (4)
        DateFormat japaneseFormat = new SimpleDateFormat("GGGy年M月d日", new Locale("ja", "JP", "JP"));
        try {
            item.setDate(japaneseFormat.parse(fieldSet.readString("date")));
        } catch (ParseException e) {
            // omitted exception handling
        }

        // (5)
        item.setCustomerId(fieldSet.readString("customerId"));

        // (6)
        DecimalFormat decimalFormat = new DecimalFormat();
        decimalFormat.setParseBigDecimal(true);
        try {
            item.setAmount((BigDecimal) decimalFormat.parse(fieldSet.readString("amount")));
        } catch (ParseException e) {
            // omitted exception handling
        }

        return item; // (7)
    }
}
```

## 設定内容の項目一覧

項番	説明
(1)	FieldSetMapperクラスを実装し、mapFieldSetメソッドをオーバーライドする。FieldSetMapperの型引数には変換対象クラスを設定する。

項目番	説明
(2)	変換処理等を行ったデータを格納するために変換対象クラスの変数を定義する。
(3)	引数で受けたFieldSetからbranchIdを取得し、変換対象クラスの変数へ格納する。 branchIdは変換処理が不要であるため、変換処理等は行っていない。
(4)	引数で受けたFieldSetからdateを取得し、変換対象クラスの変数へ格納する。 和暦フォーマットの日付をDate型へ変換するため、SimpleDateFormatでフォーマットを指定している。
(5)	引数で受けたFieldSetからcustomerIdを取得し、変換対象クラスの変数へ格納する。 customerIdは変換処理が不要であるため、変換処理等は行っていない。
(4)	引数で受けたFieldSetからamountを取得し、変換対象クラスの変数へ格納する。 カンマを含む値をBigDecimal型へ変換するため、DecimalFormatを使用している。
(7)	処理結果を保持している変換対象クラスを返す。

#### FieldSetクラスからの値取得

FieldSetクラスは、下記のような格納された値を取得するための様々なデータ型に対応したメソッドをもつ。

また、FieldSet生成時にフィールドの名前と関連付けられてデータを格納した場合は、名前指定でのデータ取得、名前を指定しない場合ではインデックスを指定してのデータ取得が可能である。



- `readString()`
- `readInt()`
- `readBigDecimal()`

など

### 5.3.3.2. XMLファイル

XMLファイルを扱う場合の定義方法を説明する。

BeanとXML間の変換処理(O/X (Object/XML) マッピング)にはSpring Frameworkが提供するライブラリを使用する。

XMLファイルとオブジェクト間の変換処理を行うライブラリとして、XStreamやJAXBなどを利用したMarshallerおよびUnmarshallerを実装クラスが提供されている。

状況に応じて適しているものを使用すること。

JAXBとXStreamを例に特徴と採用する際のポイントを説明する。

#### JAXB

- 変換対象のBeanはBean定義にて指定する
- スキーマファイルを用いたバリデーションを行うことができる
- 対外的にスキーマを定義しており、入力ファイルの仕様が厳密に決まっている場合に有用である

#### XStream

- Bean定義にて柔軟にXMLの要素とBeanのフィールドをマッピングすることができる

- 柔軟にBeanマッピングする必要がある場合に有用である

ここでは、JAXBを利用する例を示す。

#### 5.3.3.2.1. 入力

XMLファイルの入力にはSpring Batchが提供する`org.springframework.batch.item.xml.StaxEventItemReader`を使用する。`StaxEventItemReader`は指定した`Unmarshaller`を使用してXMLファイルをBeanにマッピングすることでXMLファイルを読み込むことができる。

`StaxEventItemReader`は以下の要領で定義する。

- XMLのルートエレメントとなる変換対象クラスに`@XmlRootElement`を付与する
- `StaxEventItemReader`に以下のプロパティを設定する
  - プロパティ`resource`に読み込み対象ファイルを設定する
  - プロパティ`fragmentRootElementName`にルートエレメントとなる要素の名前を設定する
  - プロパティ`unmarshaller`に`org.springframework.oxm.jaxb.Jaxb2Marshaller`を設定する
- `Jaxb2Marshaller`には以下のプロパティを設定する
  - プロパティ`classesToBeBound`に変換対象のクラスをリスト形式で設定する
  - スキーマファイルを用いたバリデーションを行う場合は、以下に示す2つのプロパティを設定する
    - プロパティ`schema`にバリデーションにて使用するスキーマファイルを設定する
    - プロパティ`validationEventHandler`にバリデーションにて発生したイベントを処理する`ValidationEventHandler`の実装クラスを設定する

下記の入力ファイルを読み込むための設定例を示す。

## 入力ファイル例

```
<?xml version="1.0" encoding="UTF-8"?>
<records>
  <SalesPlanDetail>
    <branchId>000001</branchId>
    <year>2016</year>
    <month>1</month>
    <customerId>0000000001</customerId>
    <amount>1000000000</amount>
  </SalesPlanDetail>
  <SalesPlanDetail>
    <branchId>000002</branchId>
    <year>2017</year>
    <month>2</month>
    <customerId>0000000002</customerId>
    <amount>2000000000</amount>
  </SalesPlanDetail>
  <SalesPlanDetail>
    <branchId>000003</branchId>
    <year>2018</year>
    <month>3</month>
    <customerId>0000000003</customerId>
    <amount>3000000000</amount>
  </SalesPlanDetail>
</records>
```

## 変換対象クラス

```
@XmlRootElement(name = "SalesPlanDetail") // (1)
public class SalesPlanDetailToJaxb {

  private String branchId;
  private int year;
  private int month;
  private String customerId;
  private BigDecimal amount;

  // omitted getter/setter
}
```

## 設定内容の項目一覧

項目番	説明
(1)	XML のルートタグとするため@XmlRootElementアノテーションを付与する。タグの名前にSalesPlanDetailを設定する。

上記のファイルを読む込むための設定は以下のとおり。

## Bean定義

```
<!-- (1) (2) (3) -->
<bean id="reader"
    class="org.springframework.batch.item.xml.StaxEventItemReader" scope="step"
    p:resource="file:#{jobParameters[inputFile]}"
    p:fragmentRootElementName="SalesPlanDetail"
    p:strict="true">
    <property name="unmarshaller" > <!-- (4) -->
        <!-- (5) (6) -->
        <bean class="org.springframework.oxm.jaxb.Jaxb2Marshaller"
            p:schema="file:files/test/input/ch05/fileaccess/SalesPlanDetail.xsd"
            p:validationEventHandler-ref="salesPlanDetailValidationEventHandler">
            <property name="classesToBeBound"> <!-- (7) -->
                <list>
                    <value>org.terasoluna.batch.functionalttest.ch05.fileaccess.model.plan.SalesPlanDetailT
oJaxb</value>
                </list>
            </property>
        </bean>
    </property>
</bean>
```

## 設定内容の項目一覧

項目番号	プロパティ名	設定内容	必須	デフォルト値
(1)	resource	入力ファイルを設定する。	✓	なし
(2)	fragmentRootElementName	ルートエレメントとなる要素の名前を設定する。 対象となるオブジェクトが複数ある場合には、 <code>fragmentRootElementNames</code> を利用する。		なし
(3)	strict	trueを設定すると、入力ファイルが存在しない(開けない)場合に例外が発生する。		true
(4)	unmarshaller	アンマーシャラを設定する。 JAXBを利用する場合は、 <code>org.springframework.oxm.jaxb.Jaxb2Marshaller</code> のBeanを設定する。	✓	なし
(5)	schema	バリデーションにて使用するスキーマファイルを設定する。		
(6)	validationEventHandler	バリデーションにて発生したイベントを処理する <code>ValidationEventHandler</code> の実装クラスを設定する。 <code>ValidationEventHandler</code> の実装例は後述する。		
(7)	classesToBeBound	変換対象のクラスをリスト形式で設定する。	✓	なし

## ValidationEventHandlerの実装例

```
@Component
// (1)
public class SalesPlanDetailValidationEventHandler implements ValidationEventHandler {
    /**
     * Logger.
     */
    private static final Logger logger =
        LoggerFactory.getLogger(SalesPlanDetailValidationEventHandler.class);

    @Override
    public boolean handleEvent(ValidationEvent event) {
        // (2)
        logger.error("[EVENT [SEVERITY:{}] [MESSAGE:{}] [LINKED EXCEPTION:{}]]" +
                    " [LOCATOR: [LINE NUMBER:{}] [COLUMN NUMBER:{}] [OFFSET:{}]]" +
                    " [OBJECT:{}] [NODE:{}] [URL:{}] ]",
                    event.getSeverity(),
                    event.getMessage(),
                    event.getLinkedException(),
                    event.getLocator().getLineNumber(),
                    event.getLocator().getColumnNumber(),
                    event.getLocator().getOffset(),
                    event.getLocator().getObject(),
                    event.getLocator().getNode(),
                    event.getLocator().getURL());
        return false; // (3)
    }
}
```

## 設定内容の項目一覧

項番	説明
(1)	ValidationEventHandlerクラスを実装し、handleEventメソッドをオーバーライドする。
(2)	引数で受けたevent(ValidationEvent)からイベントの情報を取得し、適宜処理を行う。 例ではイベントの情報をログ出力している。
(3)	検証処理を終了させるためfalseを返す。 検証処理を続行する場合はtrueを返す。 適切なUnmarshalException、ValidationException、またはMarshalExceptionを生成して現在の操作を終了させる場合はfalseを返す。

依存ライブラリの追加

`org.springframework.oxm.jaxb.Jaxb2Marshaller`など、Spring Frameworkが提供するライブラリであるSpring Object/XML Marshallingを使用する場合は、ライブラリの依存関係に以下の設定を追加する必要がある。



```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-oxm</artifactId>
</dependency>
```

#### 5.3.3.2.2. 出力

XMLファイルの出力にはSpring Batchが提供する`org.springframework.batch.item.xml.StaxEventItemWriter`を使用する。`StaxEventItemWriter`は指定した`Marshaller`を使用してBeanをXMLにマッピングすることでXMLファイルを出力することができる。

`StaxEventItemWriter`は以下の要領で定義する。

- 変換対象クラスに以下の設定を行う
  - XMLのルートエレメントとなるためクラスに`@XmlRootElement`を付与する
  - `@XmlAttribute`アノテーションを使用してフィールドを出力する順番を設定する
  - XMLへの変換対象外とするフィールドがある場合、対象フィールドのgetterに`@XmlTransient`アノテーションを付与する
- `StaxEventItemWriter`に以下のプロパティを設定する
  - プロパティ`resource`に出力対象ファイルを設定する
  - プロパティ`marshaller`に`org.springframework.oxm.jaxb.Jaxb2Marshaller`を設定する
- `Jaxb2Marshaller`には以下のプロパティを設定する
  - プロパティ`classesToBeBound`に変換対象のクラスをリスト形式で設定する

下記の出力ファイルを書き出すための設定例を示す。

## 出力ファイル例

```
<?xml version="1.0" encoding="UTF-8"?>
<records>
  <Customer>
    <customerId>001</customerId>
    <customerName>CustomerName001</customerName>
    <customerAddress>CustomerAddress001</customerAddress>
    <customerTel>111111111111</customerTel>
    <chargeBranchId>001</chargeBranchId></Customer>
  <Customer>
    <customerId>002</customerId>
    <customerName>CustomerName002</customerName>
    <customerAddress>CustomerAddress002</customerAddress>
    <customerTel>111111111111</customerTel>
    <chargeBranchId>002</chargeBranchId></Customer>
  <Customer>
    <customerId>003</customerId>
    <customerName>CustomerName003</customerName>
    <customerAddress>CustomerAddress003</customerAddress>
    <customerTel>111111111111</customerTel>
    <chargeBranchId>003</chargeBranchId>
  </Customer>
</records>
```

*XML*ファイル出力時のフォーマット処理(改行およびインデント)について  
上記の出力ファイル例ではフォーマット処理(改行およびインデント)済みのXMLを例示  
しているが、実際にはフォーマットされていないファイルが出力される。

*Jaxb2Marshaller*にはXML出力時にフォーマットを行う機能があるが期待どおり動作し  
ない。

この件に関してはSpring Forumにて議論されているため、今後期待どおり動作するよ  
うになる可能性がある。

これを回避し、フォーマット済みの出力を行うためには、以下のよう  
に*marshallerProperties*に設定すればよい。

```
<property name="marshaller">
    <bean class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
        <property name="classesToBeBound">
            <!-- omitted settings -->
        </property>
        <property name="marshallerProperties">
            <map>
                <entry>
                    <key>
                        <util:constant
                            static-
field="javax.xml.bind.Marshaller.JAXB_FORMATTED_OUTPUT"/>
                    </key>
                    <value type="java.lang.Boolean">true</value>
                </entry>
            </map>
        </property>
    </bean>
</property>
```



## 変換対象クラス

```
@XmlRootElement(name = "Customer") // (2)
@XmlType(propOrder={"customerId", "customerName", "customerAddress",
    "customerTel", "chargeBranchId"}) // (2)
public class CustomerToJaxb {

    private String customerId;
    private String customerName;
    private String customerAddress;
    private String customerTel;
    private String chargeBranchId;
    private Timestamp createDate;
    private Timestamp updateDate;

    // omitted getter/setter

    @XmlTransient // (3)
    public Timestamp getCreateDate() { return createDate; }

    @XmlTransient // (3)
    public Timestamp getUpdateDate() { return updateDate; }
}
```

## 設定内容の項目一覧

項目番	説明
(1)	XML のルートタグとするため@XmlRootElement アノテーションを付与する。タグの名前にCustomerを設定している。
(2)	@XmlType アノテーションを使用してフィールドを出力する順番を設定する。
(3)	XMLへの変換対象外とするフィールドのgetterに@XmlTransient アノテーションを付与する。

上記のファイルを書き出すための設定は以下のとおり。

## Bean定義

```
<!-- (1) (2) (3) (4) (5) (6) -->
<bean id="writer"
    class="org.springframework.batch.item.xml.StaxEventItemWriter" scope="step"
    p:resource="file:#{jobParameters[outputFile]}"
    p:encoding="MS932"
    p:rootTagName="records"
    p:overwriteOutput="true"
    p:shouldDeleteIfEmpty="false"
    p:transactional="true">
    <property name="marshaller"> <!-- (7) -->
        <bean class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
            <property name="classesToBeBound"> <!-- (8) -->
                <list>
                    <value>org.terasoluna.batch.functionalttest.ch05.fileaccess.model.mst.CustomerToJaxb</value>
                </list>
            </property>
        </bean>
    </property>
</bean>
```

## 設定内容の項目一覧

項目番号	プロパティ名	設定内容	必須	デフォルト値
(1)	resource	出力ファイルを設定する	✓	なし
(2)	encoding	入力ファイルの文字コードを設定する		JavaVMのデフォルト文字セット
(3)	rootTagName	XMLのルートタグを設定する。		
(4)	overwriteOutput	trueの場合、既にファイルが存在すれば削除する。 falseの場合、既にファイルが存在すれば例外をスローする。		true
(5)	shouldDeleteIfEmpty	trueの場合、出力結果が空ファイルであれば削除する。		false
(6)	transactional	トランザクション制御を行うかを設定する。 詳細は、 <a href="#">トランザクション制御</a> を参照。		true
(7)	marshaller	マーシャラを設定する。 JAXBを利用する場合は、 <code>org.springframework.oxm.jaxb.Jaxb2Marshaller</code> を設定する。	✓	なし
(8)	classesToBeBound	変換対象のクラスをリスト形式で設定する。	✓	なし

依存ライブラリの追加

`org.springframework.oxm.jaxb.Jaxb2Marshaller`など、Spring Frameworkが提供するライブラリであるSpring Object/XML Marshallingを使用する場合は、ライブラリの依存関係に以下の設定を追加する必要がある。



```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-oxm</artifactId>
</dependency>
```

## ヘッダ・フッタの出力

ヘッダとフッタの出力には、`org.springframework.batch.item.xml.StaxWriterCallback`の実装クラスを使用する。

ヘッダの出力は、`headerCallback`、フッタの出力は、`footerCallback`に`StaxWriterCallback`の実装を設定する。

以下に出力されるファイルの例を示す。

ヘッダはルートエレメント開始タグの直後、フッタはルートエレメント終了タグの直前に出力される。

## 出力ファイル例

```
<?xml version="1.0" encoding="UTF-8"?>
<records>
<!-- Customer list header -->
<Customer>
    <customerId>001</customerId>
    <customerName>CustomerName001</customerName>
    <customerAddress>CustomerAddress001</customerAddress>
    <customerTel>111111111111</customerTel>
    <chargeBranchId>001</chargeBranchId></Customer>
<Customer>
    <customerId>002</customerId>
    <customerName>CustomerName002</customerName>
    <customerAddress>CustomerAddress002</customerAddress>
    <customerTel>111111111111</customerTel>
    <chargeBranchId>002</chargeBranchId></Customer>
<Customer>
    <customerId>003</customerId>
    <customerName>CustomerName003</customerName>
    <customerAddress>CustomerAddress003</customerAddress>
    <customerTel>111111111111</customerTel>
    <chargeBranchId>003</chargeBranchId>
</Customer>
<!-- Customer list footer -->
</records>
```

XMLファイル出力時のフォーマット処理(改行およびインデント)について

上記の出力ファイル例ではフォーマット処理(改行およびインデント)済みのXMLを例示しているが、実際にはフォーマットされていないファイルが出力される。



詳細は、[XMLファイル出力時のフォーマット処理\(改行およびインデント\)について](#)を参照すること。

上記のようなファイルを出力する設定を以下に示す。

#### Bean定義

```
<!-- (1) (2) -->
<bean id="writer"
    class="org.springframework.batch.item.xml.StaxEventItemWriter" scope="step"
    p:resource="file:${jobParameters[outputFile]}"
    p:headerCallback-ref="writeHeaderStaxWriterCallback"
    p:footerCallback-ref="writeFooterStaxWriterCallback">
    <property name="marshaller">
        <!-- omitted settings -->
    </property>
</bean>
```

#### 設定内容の項目一覧

項目番	プロパティ名	設定内容	必須	デフォルト値
(1)	headerCallback	StaxWriterCallbackの実装クラスを設定する 。		
(2)	footerCallback	StaxWriterCallbackの実装クラスを設定する 。		

StaxWriterCallbackは以下の要領で実装する。

- StaxWriterCallbackクラスを実装し、writeメソッドをオーバーライドする
- 引数で受けるXMLEventWriterを用いてヘッダ/フッタを出力する

## StaxWriterCallbackの実装例

```
@Component
public class WriteHeaderStaxWriterCallback implements StaxWriterCallback { // (1)
    @Override
    public void write(XMLEventWriter writer) throws IOException {
        XMLEventFactory factory = XMLEventFactory.newInstance();
        try {
            writer.add(factory.createComment(" Customer list header ")); // (2)
        } catch (XMLStreamException e) {
            // omitted exception handling
        }
    }
}
```

### 設定内容の項目一覧

項番	説明
(1)	StaxWriterCallbackクラスを実装し、writeメソッドをオーバーライドする。
(2)	引数で受けるXMLEventWriterを用いてヘッダ/フッタを出力する。

XMLEventFactoryを使用したXMLの出力

XMLEventWriterクラスを用いたXMLファイルの出力ではXMLEventFactoryクラスを使用することで効率的にXMLEventを生成することができる。



XMLEventWriterクラスにはaddメソッドが定義されており、XMLEventオブジェクトを引数に取りXMLファイルの出力を行う。

XMLEventオブジェクトを都度生成するのは非常に手間が掛かるため、XMLEventを容易に生成することができるXMLEventFactoryクラスを使用する。

XMLEventFactoryクラスにはcreateStartDocumentメソッドやcreateStartElementメソッドなど、作成するイベントに対応したメソッドが定義してある。

### 5.3.3.3. マルチフォーマット

マルチフォーマットファイルを扱う場合の定義方法を説明する。

マルチフォーマットは、Overviewで説明したとおり(ヘッダn行 + データn行 + トレーラn行)\* n + フッタn行 の形式を基本とするが以下ののようなパターンも存在する。

- ・ フッタレコードがある場合、ない場合
- ・ 同一レコード区分内でフォーマットが異なるレコードがある場合
  - ・ 例)データ部は項目数が5と6のデータレコードが混在する

マルチフォーマットのパターンはいくつかあるが、実現方式は同じになる。

### 5.3.3.3.1. 入力

マルチフォーマットファイルの読み込みには、Spring Batchが提供する`org.springframework.batch.item.file.mapping.PatternMatchingCompositeLineMapper`を使用する。マルチフォーマットファイルでは各レコードのフォーマットごとに異なるBeanにマッピングする必要がある。

`PatternMatchingCompositeLineMapper`は、正規表現によってレコードに対して使用する`LineTokenizer` および`FieldSetMapper` を選択することができる。

たとえば、以下のような形で使用する`LineTokenizers` を選択することが可能である。

- 正規表現`USER*`(レコードの先頭が`USER`である)にマッチする場合は`userTokenizer`を使用する
- 正規表現`LINEA*`(レコードの先頭が`LINEA`である)にマッチする場合は`lineATokenizer`を使用する



マルチフォーマットファイルを読み込む際のレコードにかかるフォーマットの制約

マルチフォーマットファイルの読み込みを行うためには、正規表現でレコード区分を判別可能なフォーマットでなければならない。

`PatternMatchingCompositeLineMapper`は以下の要領で実装する。

- 変換対象クラスはレコード区分をもつクラスを定義し、各レコード区分のクラスに継承させる
- 各レコードをBeanにマッピングするための`LineTokenizer`および`FieldSetMapper`を定義する
- `PatternMatchingCompositeLineMapper`を定義する
  - プロパティ`tokenizers`に各レコード区分に対応する`LineTokenizer`を設定する
  - プロパティ`fieldSetMappers`に各レコード区分に対応する`FieldSetMapper`を設定する

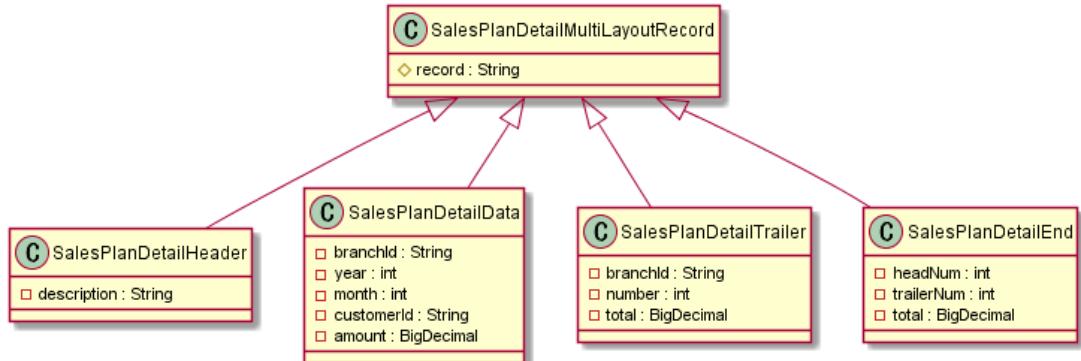
変換対象クラスはレコード区分をもつクラスを定義し、各レコード区分のクラスに継承させる

`ItemProcessor`は1つの型を引数に取る仕様である。

しかし、単純に`PatternMatchingCompositeLineMapper`にてマルチフォーマットのファイルをレコード区分ごとに異なるBeanにマッピングすると、`ItemProcessor`は1つの型を引数に取るため複数の型を処理することができない。

そのため、変換対象のクラスに継承関係をもたせ、`ItemProcessor`の引数の型にスーパークラスを指定することで解決が可能である。

以下に変換対象クラスのクラス図と`ItemProcessor`の定義例を示す。



変換対象クラスのクラス図

#### ItemProcessorの定義例

```

public class MultiLayoutItemProcessor implements
    ItemProcessor<SalesPlanDetailMultiLayoutRecord, String> {
    @Override
    // (1)
    public String process(SalesPlanDetailMultiLayoutRecord item)
    throws Exception {
        String record = item.getRecord(); // (2)

        switch (record) { // (3)
            case "H":
                // omitted business logic
            case "D":
                // omitted business logic
            case "T":
                // omitted business logic
            case "E":
                // omitted business logic
            default:
                // omitted exception handling
        }
    }
}

```

#### 設定内容の項目一覧

項目番	説明
(1)	ItemProcessorの引数に継承関係をもたせた変換対象クラスのスーパークラスを設定する。
(2)	itemからレコード区分を取得する。 各レコード区分によって実態のクラスは異なるが、ポリモーフィズムによってレコード区分を取得できる。
(3)	レコード区分を判定し、各レコード区分ごとの処理を行う。 適宜、クラスの変換処理等を行うこと。

以下に下記の入力ファイルを読み込むための設定例を示す。実装例を示す。

## 入力ファイル例

```
H,Sales_plan_detail header No.1
D,000001,2016,1,0000000001,100000000
D,000001,2016,1,0000000002,200000000
D,000001,2016,1,0000000003,300000000
T,000001,3,600000000
H,Sales_plan_detail header No.2
D,00002,2016,1,0000000004,400000000
D,00002,2016,1,0000000005,500000000
D,00002,2016,1,0000000006,600000000
T,00002,3,1500000000
H,Sales_plan_detail header No.3
D,00003,2016,1,0000000007,700000000
D,00003,2016,1,0000000008,800000000
D,00003,2016,1,0000000009,900000000
T,00003,3,2400000000
E,3,9,4500000000
```

下記に変換対象クラスのBean定義例を示す。

## 変換対象クラス

```
/**
 * Model of record indicator of sales plan detail.
 */
public class SalesPlanDetailMultiLayoutRecord {

    protected String record;

    // omitted getter/setter
}

/**
 * Model of sales plan detail header.
 */
public class SalesPlanDetailHeader extends SalesPlanDetailMultiLayoutRecord {

    private String description;

    // omitted getter/setter
}

/**
 * Model of Sales plan Detail.
 */
public class SalesPlanDetailData extends SalesPlanDetailMultiLayoutRecord {

    private String branchId;
    private int year;
}
```

```

private int month;
private String customerId;
private BigDecimal amount;

// omitted getter/setter
}

/** 
 * Model of Sales plan Detail.
 */
public class SalesPlanDetailTrailer extends SalesPlanDetailMultiLayoutRecord {

    private String branchId;
    private int number;
    private BigDecimal total;

    // omitted getter/setter
}

/** 
 * Model of Sales plan Detail.
 */
public class SalesPlanDetailEnd extends SalesPlanDetailMultiLayoutRecord {
    // omitted getter/setter

    private int headNum;
    private int trailerNum;
    private BigDecimal total;

    // omitted getter/setter
}

```

上記のファイルを読む込むための設定は以下のとおり。

#### Bean定義例

```

<!-- (1) -->
<bean id="headerDelimitedLineTokenizer"
      class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
      p:names="record,description"/>

<bean id="dataDelimitedLineTokenizer"
      class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
      p:names="record,branchId,year,month,customerId,amount"/>

<bean id="trailerDelimitedLineTokenizer"
      class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
      p:names="record,branchId,number,total"/>

<bean id="endDelimitedLineTokenizer"
      class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"

```

```

    p:names="record,headNum,trailerNum,total"/>

    <!-- (2) -->
<bean id="headerBeanWrapperFieldSetMapper"
      class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"

    p:targetType="org.terasoluna.batch.functionalttest.ch05.fileaccess.model.plan.SalesPlan
DetailHeader"/>

<bean id="dataBeanWrapperFieldSetMapper"
      class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"

    p:targetType="org.terasoluna.batch.functionalttest.ch05.fileaccess.model.plan.SalesPlan
DetailData"/>

<bean id="trailerBeanWrapperFieldSetMapper"
      class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"

    p:targetType="org.terasoluna.batch.functionalttest.ch05.fileaccess.model.plan.SalesPlan
DetailTrailer"/>

<bean id="endBeanWrapperFieldSetMapper"
      class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"

    p:targetType="org.terasoluna.batch.functionalttest.ch05.fileaccess.model.plan.SalesPlan
DetailEnd"/>

<bean id="reader"
      class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
      p:resource="#{jobParameters[inputFile]}>
        <property name="lineMapper"> <!-- (3) -->
          <bean
            class="org.springframework.batch.item.file.mapping.PatternMatchingCompositeLineMapper"
          >
            <property name="tokenizers"> <!-- (4) -->
              <map>
                <entry key="H*" value-ref="headerDelimitedLineTokenizer"/>
                <entry key="D*" value-ref="dataDelimitedLineTokenizer"/>
                <entry key="T*" value-ref="trailerDelimitedLineTokenizer"/>
                <entry key="E*" value-ref="endDelimitedLineTokenizer"/>
              </map>
            </property>
            <property name="fieldSetMappers"> <!-- (5) -->
              <map>
                <entry key="H*" value-ref="headerBeanWrapperFieldSetMapper"/>
                <entry key="D*" value-ref="dataBeanWrapperFieldSetMapper"/>
                <entry key="T*" value-ref="trailerBeanWrapperFieldSetMapper"/>
                <entry key="E*" value-ref="endBeanWrapperFieldSetMapper"/>
              </map>
            </property>
          </bean>
        </property>
      </bean>
    </property>
  </bean>

```

```
</property>
</bean>
```

## 設定内容の項目一覧

項目番	プロパティ名	設定内容	必須	デフォルト値
(1)	各レコードに対応するLineTokenizer	各レコード区分に対応するLineTokenizerを定義する。		
(2)	各レコードに対応するFieldSetMapper	各レコード区分に対応するFieldSetMapperを定義する。		
(3)	lineMapper	org.springframework.batch.item.file.mapping.PatternMatchingCompositeLineMapperを設定する。	✓	なし
(3)	tokenizers	map形式で各レコード区分に対応するLineTokenizerを設定する。 keyにレコードを判別する正規表現を設定し、value-refに使用するLineTokenizerを設定する。	✓	なし
(4)	tokenizers	map形式で各レコード区分に対応するFieldSetMapperを設定する。 keyにレコードを判別する正規表現を設定し、value-refに使用するFieldSetMapperを設定する。	✓	なし

### 5.3.3.3.2. 出力

マルチフォーマットファイルを扱う場合の定義方法を説明する。

マルチフォーマットファイル読み込みではレコード区分によって使用するLineTokenizerおよびFieldSetMapperを判別するPatternMatchingCompositeLineMapperを使用することで実現可能である。しかし、書き込み時に同様の機能をもつコンポーネントは提供されていない。

そのため、ItemProcessor内で変換対象クラスをレコード(文字列)に変換する処理までを行い、ItemWriterでは受け取った文字列をそのまま書き込みを行うことでマルチフォーマットファイルの書き込みを実現する。

マルチフォーマットファイルの書き込みは以下の要領で実装する。

- ItemProcessorにて変換対象クラスをレコード(文字列)に変換してItemWriterに渡す
  - 例では、各レコード区分ごとのLineAggregatorおよびFieldExtractorを定義し、ItemProcessorでインジェクトして使用する
- ItemWriterでは受け取った文字列をそのままファイルへ書き込みを行う
  - ItemWriterのプロパティlineAggregatorにPassThroughLineAggregatorを設定する
  - PassThroughLineAggregatorは受け取ったitemのitem.toString()した結果を返すLineAggregatorである

以下に下記の出力ファイルを読み込むための設定例を示す。実装例を示す。

## 出力ファイル例

```
H,Sales_plan_detail header No.1  
D,00001,2016,1,0000000001,100000000  
D,00001,2016,1,0000000002,200000000  
D,00001,2016,1,0000000003,300000000  
T,00001,3,600000000  
H,Sales_plan_detail header No.2  
D,00002,2016,1,0000000004,400000000  
D,00002,2016,1,0000000005,500000000  
D,00002,2016,1,0000000006,600000000  
T,00002,3,1500000000  
H,Sales_plan_detail header No.3  
D,00003,2016,1,0000000007,700000000  
D,00003,2016,1,0000000008,800000000  
D,00003,2016,1,0000000009,900000000  
T,00003,3,2400000000  
E,3,9,4500000000
```

変換対象クラスの定義および[ItemProcessor](#)定義例、注意点はマルチフォーマットの入力と同様である。

上記のファイルを出力するための設定は以下のとおり。 [ItemProcessor](#)の定義例をBean定義例の後に示す。

## Bean定義例

```
<!-- (1) -->
<bean id="headerDelimitedLineAggregator"
      class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
    <property name="fieldExtractor">
      <bean
        class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"
          p:names="record,description"/>
    </property>
</bean>

<bean id="dataDelimitedLineAggregator"
      class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
    <property name="fieldExtractor">
      <bean
        class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"
          p:names="record,branchId,year,month,customerId,amount"/>
    </property>
</bean>

<bean id="trailerDelimitedLineAggregator"
      class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
    <property name="fieldExtractor">
      <bean
        class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"
          p:names="record,branchId,number,total"/>
    </property>
</bean>

<bean id="endDelimitedLineAggregator"
      class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
    <property name="fieldExtractor">
      <bean
        class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"
          p:names="record,headNum,trailerNum,total"/>
    </property>
</bean>

<bean id="writer" class="org.springframework.batch.item.file.FlatFileItemWriter"
scope="step"
      p:resource="file:#{jobParameters[outputFile]}"/>
    <property name="lineAggregator" > <!-- (2) -->
      <bean
        class="org.springframework.batch.item.file.transform.PassThroughLineAggregator"/>
    </property>
</bean>
```

## 設定内容の項目一覧

項目番	プロパティ名	設定内容	必須	デフォルト値
(1)	各レコード区分に対応するLineAggregatorおよびFieldExtractorを定義する。 ItemProcessorでLineAggregatorをインジェクトして使用する。			
(2)	lineAggregator	org.springframework.batch.item.file.transform.PassThroughLineAggregatorを設定する。 。	✓	なし

ItemProcessorの実装例を以下に示す。

例で実装しているのは、受け取ったitemを文字列に変換してItemWriterに渡す処理のみである。

## ItemProcessorの定義例

```
public class MultiLayoutItemProcessor implements
    ItemProcessor<SalesPlanDetailMultiLayoutRecord, String> {

    // (1)
    @Inject
    @Named("headerDelimitedLineAggregator")
    DelimitedLineAggregator<SalesPlanDetailMultiLayoutRecord>
    headerDelimitedLineAggregator;

    @Inject
    @Named("dataDelimitedLineAggregator")
    DelimitedLineAggregator<SalesPlanDetailMultiLayoutRecord>
    dataDelimitedLineAggregator;

    @Inject
    @Named("trailerDelimitedLineAggregator")
    DelimitedLineAggregator<SalesPlanDetailMultiLayoutRecord>
    trailerDelimitedLineAggregator;

    @Inject
    @Named("endDelimitedLineAggregator")
    DelimitedLineAggregator<SalesPlanDetailMultiLayoutRecord>
    endDelimitedLineAggregator;

    @Override
    // (2)
    public String process(SalesPlanDetailMultiLayoutRecord item) throws Exception {
        String record = item.getRecord(); // (3)

        switch (record) { // (4)
            case "H":
                return headerDelimitedLineAggregator.aggregate(item); // (5)
            case "D":
                return dataDelimitedLineAggregator.aggregate(item); // (5)
            case "T":
                return trailerDelimitedLineAggregator.aggregate(item); // (5)
            case "E":
                return endDelimitedLineAggregator.aggregate(item); // (5)
            default:
                throw new IncorrectRecordClassificationException(
                    "Record classification is incorrect.[value:" + record + "]");
        }
    }
}
```

## 設定内容の項目一覧

項目番	説明
(1)	各レコード区分に対応するLineAggregatorをインジェクトする。
(2)	ItemProcessorの引数に継承関係をもたせた変換対象クラスのスーパークラスを設定する。
(3)	itemからレコード区分を取得する。
(4)	レコード区分を判定し、各レコード区分ごとの処理を行う。
(5)	各レコード区分に対応するLineAggregatorを使用し変換対象クラスをレコード(文字列)に変換してItemWriterに渡す。

## 5.4. 排他制御

### 5.4.1. Overview

排他制御とは、複数のトランザクションから同じリソースに対して、同時に更新処理が行われる際に、データの整合性を保つために行う処理のことである。複数のトランザクションから同じリソースに対して、同時に更新処理が行われる可能性がある場合は、基本的に排他制御を行う必要がある。

ここでの複数トランザクションとは以下のことと指す。

- ・複数ジョブの同時実行時におけるトランザクション
- ・オンライン処理との同時実行時におけるトランザクション

#### 複数ジョブの排他制御



複数ジョブを同時実行する場合は、排他制御の必要がないようにジョブ設計を行うことが基本である。これは、アクセスするリソースや処理対象をジョブごとに分割することが基本であることを意味する。

排他制御に関する概念は、オンライン処理と同様であるため、TERASOLUNA Server 5.x 開発ガイドラインにある [排他制御](#)を参照してほしい。

ここでは、TERASOLUNA Server 5.xでは説明されていない部分を中心に説明をする。

本機能は、チャンクモデルとタスクレットモデルとで同じ使い方になる。

#### 5.4.1.1. 排他制御の必要性

排他制御の必要性に関しては、TERASOLUNA Server 5.x 開発ガイドラインにある [排他制御の必要性](#)を参照。

#### 5.4.1.2. ファイルの排他制御

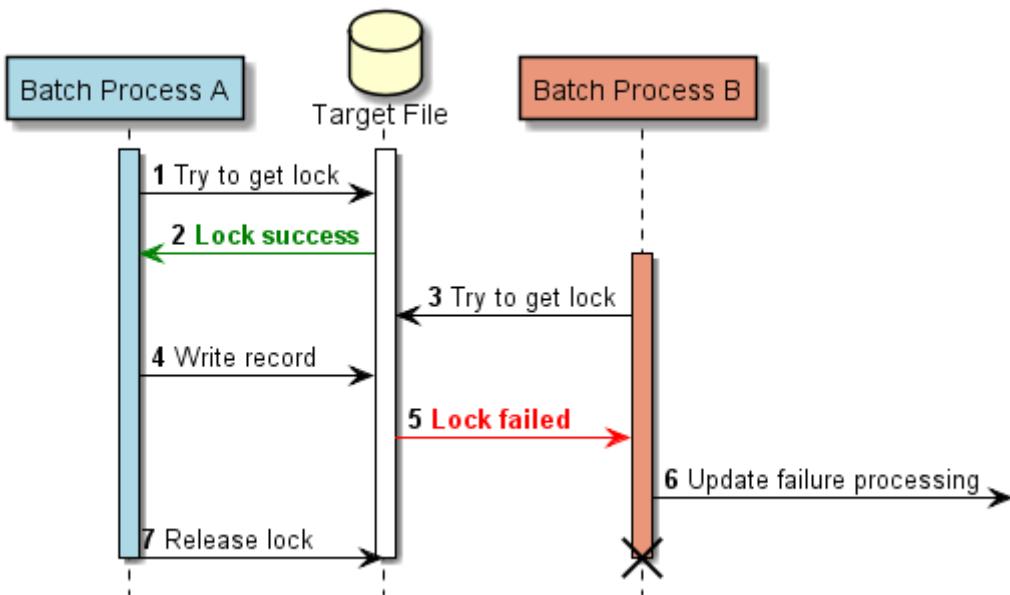
ファイルでの排他制御はファイルロックにより実現するのが一般的である。

ファイルロックとは

ファイルロックとは、ファイルのあるプログラムで使用している間、ほかのプログラムからの読み書きを制限する仕組みである。ファイルロックの処理概要を以下に示す。

シナリオ

- ・バッチ処理Aがファイルのロックを取得し、ファイルの更新処理を開始する。
- ・バッチ処理Bが同一のファイルの更新を試みファイルのロック取得を試みるが失敗する。
- ・バッチ処理Aが処理を終了し、ファイルのロックを解除する



### ファイルロックの処理概要

1. バッチ処理A(Batch ProcessA)が対象ファイル(TargetFile)のロック取得を試みる。
2. バッチ処理Aが、対象ファイルのロック取得に成功する
3. バッチ処理B(Batch ProcessB)が、対象ファイルのロック取得を試みる
4. バッチ処理Aが、対象ファイルに書き込みを行う
5. バッチ処理Bは、バッチ処理Aがロック中であるため、対象ファイルのロック取得に失敗する
6. バッチ処理Bが、ファイル更新失敗の処理を行う。
7. バッチ処理Aが、対象ファイルのロックを開放する。

#### デッドロックの予防

ファイルにおいてもデータベースと同様に複数のファイルに対してロックを取得する場合、デッドロックとなる場合がある。そのため、ファイルの更新順序をルール化することが重要である。

デッドロックの予防に関してはデータベースのテーブル間でのデッドロック防止と同様である。詳細については、TERASOLUNA Server 5.x 開発ガイドラインの [デッドロックの予防](#) を参照。



### 5.4.1.3. データベースの排他制御

データベースの排他制御に関しては、TERASOLUNA Server 5.x 開発ガイドラインにある [データベースのロック機能による排他制御](#) で詳しく説明されているため、そちらを参照のこと。

### 5.4.1.4. 排他制御方式の使い分け

TERASOLUNA Batch 5.xでのロック方式と向いているシチュエーションを示す。

#### 排他制御方式の使い分け

ロック方式	向いているシチュエーション
楽観ロック	同時実行時におけるトランザクションで、別トランザクションの更新結果を処理対象外にして処理を継続できる場合

ロック方式	向いているシチュエーション
悲観ロック	処理時間が長く、処理中に対象データの状況が変化したことによるやり直しが難しい処理 ファイルに対する排他制御が必要な処理

#### 5.4.1.5. 排他制御とコンポーネントの関係

TERASOLUNA Batch 5.xが提供する各コンポーネントと排他制御との関係は以下のとおり。

##### 楽観ロック

###### 排他制御とコンポーネントの関係

処理モデル	コンポーネント	ファイル	データベース
チャンク	ItemReader	-	Versionカラムなど取得時と更新時とで同じデータであることが確認できるカラムを含めてデータ取得を行う。
	ItemProcessor	-	排他制御は不要である。
	ItemWriter	-	取得時と更新時との差分を確認し、他の処理で更新されていないことを確認した上で更新を行う。
タスクレット	Tasklet	-	データ取得時にはItemReader、データ更新時はItemWriterで説明した処理を実施する。 Mapperインターフェースを直接利用する場合も考え方は同じである。



###### ファイルに対する楽観ロック

ファイルの特性上、ファイルに対して楽観ロックを適用することがない。

##### 悲観ロック

###### 排他制御とコンポーネントの関係

処理モデル	コンポーネント	ファイル	データベース
チャンク	ItemReader	-	SQL文のFOR UPDATEを利用する。
	ItemProcessor		ロックされたデータを扱うのが基本であるため、ここでは原則排他制御は行わない。
	ItemWriter	-	排他を意識することなくデータを更新する。
タスクレット	Tasklet	ItemStreamReaderでファイルをオープンした直後にファイルロックを取得する。 ItemStreamWriterをクローズする直前にファイルロックを開放する。	データ取得時にはItemReader、データ更新時はItemWriterで説明した処理を実施する。 Mapperインターフェースを直接利用する場合も考え方は同じである。

ファイルに対する悲観ロック



ファイルに対する悲観ロックはタスクレットモデルで実装すること。チャンクモデルではその構造上、チャンク処理の隙間で排他できない期間が存在してしまうためである。また、ファイルアクセスはItemStreamReader/ItemStreamWriterをInjectして利用することを前提とする。

データベースでの悲観ロックによる待ち時間



悲観ロックを行う場合、競合により処理が待たされる時間が長くなる可能性がある。その場合、NO WAITオプションやタイムアウト時間を指定して、悲観ロックを使用するのが妥当である。

## 5.4.2. How to use

排他制御の使い方をリソース別に説明する。

- ファイルの排他制御
- データベースの排他制御

### 5.4.2.1. ファイルの排他制御

TERASOLUNA Batch 5.xにおけるファイルの排他制御はタスクレットを実装することで実現する。排他の実現手段としては、`java.nio.channels.FileChannel`クラスを使用したファイルロック取得で排他制御を行う。



*FileChannel*クラスの詳細

`FileChannel`クラスの詳細、使用方法については[Javadoc](#)を参照。

`FileChannel`クラスを使用しファイルのロックを取得する例を示す。

Tasklet実装

```
@Component
@Scope("step")
public class FileExclusiveTasklet implements Tasklet {

    private String targetPath = null; // (1)

    @Inject
    ItemStreamReader<SalesPlanDetail> reader;

    @Inject
    ItemStreamWriter<SalesPlanDetailWithProcessName> writer;

    @Override
    public RepeatStatus execute(StepContribution contribution,
        ChunkContext chunkContext) throws Exception {

        // omitted.
```

```

FileChannel fc = null;
FileLock fileLock = null;

try {
    reader.open(executionConetxt);
    writer.open(executionConetxt); // (2)

    try {
        File file = new File(targetPath);
        fc = FileChannel.open(file.toPath(), StandardOpenOption.WRITE,
            StandardOpenOption.CREATE,
            StandardOpenOption.APPEND); // (3)
        fileLock = fc.tryLock(); // (4)
    } catch (IOException e) {
        logger.error("Failure other than lock acquisition", e);
        throw new FailedOtherAcquireLockException(
            "Failure other than lock acquisition", e);
    }
    if (fileLock == null) {
        logger.error("Failed to acquire lock. [processName={}]", processName);
        throw new FailedAcquireLockException("Failed to acquire lock");
    }

    // (5)
    SalesPlanDetail item;
    List<SalesPlanDetailWithProcessName> items = new ArrayList<>();
    while ((item = reader.read()) != null) {

        // omitted.

        items.add(item);
        if (items.size() >= 10) {
            writer.write(items);
            items.clear();
        }
    }
    if (items.size() > 0) {
        writer.write(items);
    }

} finally {
    if (fileLock != null) {
        try {
            fileLock.release(); // (6)
        } catch (IOException e) {
            logger.warn("Lock release failed.", e);
        }
    }
    if (fc != null) {
        try {

```

```

        fc.close();
    } catch (IOException e) {
        // ignore
    }
}
writer.close(); // (7)
reader.close();
}
return RepeatStatus.FINISHED;
}

// (8)
@Value("${jobParameters[outputFile]$")
public void setTargetPath(String targetPath) {
    this.targetPath = targetPath;
}
}

```

## 説明

項目番号	説明
(1)	排他対象のファイルパス。
(2)	排他対象のファイルをオープンする。
(3)	ファイルチャネルを取得する。 この例では、ファイルの新規作成・追記・書き込みに対するチャネルを取得している。
(4)	ファイルロックを取得する。
(5)	ファイル出力を伴うビジネスロジックを実行する。
(6)	ファイルロックを開放する。
(7)	排他対象のファイルをクローズする。
(8)	ファイルパスを設定する。 この例では、ジョブパラメータから受け取るようにしている。

ロック取得に用いる*FileChannel*のメソッドについて



`lock()`メソッドは対象ファイルがロック済みの場合ロックが解除されるまで待機するため、待機されない`tryLock()`メソッドを使用することを推奨する。なお`tryLock()`は共有ロックと排他ロックが選択できるが、バッチ処理においては、通常は排他ロックを用いる。

### 同一VMでのスレッド間の排他制御

同一VMにおけるスレッド間の排他制御は注意が必要である。同一VMでのスレッド間でファイルに対する処理を行う場合、`FileChannel`クラスを用いたのロック機能では、ファイルが別スレッドの処理にてロックされているかの判定ができない。



そのため、スレッド間での排他制御は機能しない。これを回避するには、ファイルへの書き込みを行う部分で同期化処理をすることでスレッド間の排他制御が行える。

しかし、同期化を行うことで並列処理のメリットが薄れてしまい、単一スレッドで処理することと差異がなくなってしまう。結果、同一のファイルに対して異なるスレッドで排他制御をして処理することは適していないため、そのような処理設計・実装を行わないこと。

### *FlatFileItemWriter*のappendAllowedプロパティについて



ファイルを新規作成(上書き)する場合は、`appendAllowed`プロパティを`false`(デフォルト)にすることで、排他制御が実現できる。これは、`FlatFileItemWriter`の内部で`FileChannel`を使って制御しているためである。しかし、ファイルの追記(`appendAllowed`プロパティが`true`)の場合は、開発者が`FileChannel`による排他制御を実装する必要がある。

#### 5.4.2.2. データベースの排他制御

TERASOLUNA Batch 5.xにおけるデータベースの排他制御について説明する。

データベースの排他制御実装は、TERASOLUNA Server 5.x 開発ガイドラインにある [MyBatis3使用時の実装方法](#)が基本である。本ガイドラインでは、[MyBatis3使用時の実装方法](#)ができている前提で説明を行う。

[排他制御とコンポーネントの関係](#)にあるとおり、処理モデル・コンポーネントの組み合わせによるバリエーションがある。

##### データベースの排他制御のバリエーション

排他方式	処理モデル	コンポーネント
楽観ロック	チャンクモデル	ItemReader/ItemWriter
	タスクレットモデル	ItemReader/ItemWriter
		Mapperインターフェース
悲観ロック	チャンクモデル	ItemReader/ItemWriter
	タスクレットモデル	ItemReader/ItemWriter
		Mapperインターフェース

タスクレットモデルでMapperインターフェースを使用する場合は、[MyBatis3使用時の実装方法](#)のとおりであるため、説明を割愛する。

タスクレットモデルでItemReader/ItemWriterを使用する場合は、Mapperインターフェースでの呼び出し部分がItemReader/ItemWriterに代わるだけなので、これも説明を割愛する。

よって、ここではチャンクモデルの排他制御について説明する。

#### 5.4.2.2.1. 楽観ロック

チャンクモデルでの楽観ロックについて説明する。

MyBatisBatchItemWriterがもつ`assertUpdates`プロパティの設定により、ジョブの振る舞いが変化するので業務要件に合わせて、適切に設定をする必要がある。

楽観ロックを行うジョブ定義を以下に示す。

ジョブ定義

```
<!-- (1) -->
<bean id="reader"
      class="org.mybatis.spring.batch.MyBatisCursorItemReader" scope="step"

      p:queryId="org.terasoluna.batch.functionalttest.ch05.exclusivecontrol.repository.ExclusiveControlRepository.branchFindOne"
      p:sqlSessionFactory-ref="jobSqlSessionFactory"/>
      <property name="parameterValues">
          <map>
              <entry key="branchId" value="#{jobParameters[branchId]}"/>
          </map>
      </property>
  </bean>

<!-- (2) --->
<bean id="writer"
      class="org.mybatis.spring.batch.MyBatisBatchItemWriter" scope="step"

      p:statementId="org.terasoluna.batch.functionalttest.ch05.exclusivecontrol.repository.ExclusiveControlRepository.branchExclusiveUpdate"
      p:sqlSessionTemplate-ref="batchModeSqlSessionTemplate"
      p:assertUpdates="true" /> <!-- (3) -->

<batch:job id="chunkOptimisticLockCheckJob" job-repository="jobRepository">
    <batch:step id="chunkOptimisticLockCheckJob.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="reader" processor="branchEditItemProcessor"
                         writer="writer" commit-interval="10" />
        </batch:tasklet>
    </batch:step>
</batch:job>
```

説明

項番	説明
(1)	楽観ロックによるデータ取得のSQLIDを設定する。
(2)	楽観ロックによるデータ更新のSQLIDを設定する。

項目番	説明
(3)	バッチ更新の件数を検証有無を設定する。 <code>true</code> (デフォルト)に設定すると、更新件数が0件の場合に例外をスローする。 <code>false</code> に設定すると、更新件数が0件の場合でも正常処理とする。

#### 5.4.2.2. 悲観ロック

チャンクモデルでの悲観ロックについて説明する。

悲観ロックを行うジョブ定義を以下に示す。

ジョブ定義

```

<!-- (1) -->
<bean id="reader"
      class="org.mybatis.spring.batch.MyBatisCursorItemReader" scope="step"

      p:queryId="org.terasoluna.batch.functionalttest.ch05.exclusivecontrol.repository.ExclusiveControlRepository.branchFindOneWithNowWaitLock"
      p:sqlSessionFactory-ref="jobSqlSessionFactory">
    <property name="parameterValues">
      <map>
        <entry key="branchId" value="#{jobParameters[branchId]}"/>
      </map>
    </property>
</bean>

<!-- (2) -->
<bean id="writer"
      class="org.mybatis.spring.batch.MyBatisBatchItemWriter" scope="step"

      p:statementId="org.terasoluna.batch.functionalttest.ch05.exclusivecontrol.repository.ExclusiveControlRepository.branchUpdate"
      p:sqlSessionTemplate-ref="batchModeSqlSessionTemplate"
      p:assertUpdates="#{new Boolean(jobParameters[assertUpdates])} />

<batch:job id="chunkPessimisticLockCheckJob" job-repository="jobRepository">
  <batch:step id="chunkPessimisticLockCheckJob.step01">
    <batch:tasklet transaction-manager="jobTransactionManager">
      <batch:chunk reader="reader" processor="branchEditItemProcessor"
                    writer="writer" commit-interval="10" />
    </batch:tasklet>
  </batch:step>
  <batch:listeners>
    <batch:listener ref="jobExecutionLoggingListener"/>
  </batch:listeners>
</batch:job>
```

説明

項目番	説明
(1)	悲観ロックによるデータ取得のSQLIDを設定する。
(2)	排他制御をしないデータ更新のSQLと同じSQLIDを設定する。

排他された時の振る舞い



NO WAITやタイムアウトを設定して悲観ロックを行う場合、他のトランザクションにより排他される時は、MyBatisCursorItemReaderの**doOpen()**メソッドで例外が発生する。

# Chapter 6. 異常系への対応

## 6.1. 入力チェック

### 6.1.1. Overview

本節では、ジョブの入力データに対する妥当性のチェック(以降、入力チェックと呼ぶ)について説明する。

本機能は、チャングルモデルとタスクレットモデルとで同じ使い方になる。

一般的に、バッチ処理における入力チェックは、他システム等から受領したデータに対して、自システムにおいて妥当であることを確認するために実施する事が多い。

反対に、自システム内の信頼できるデータ(たとえば、データベースに格納されたデータ)に対して、入力チェックを実施することは不要と言える。

入力チェックはTERASOLUNA Server 5.xの内容と重複するため、TERASOLUNA Server 5.x 開発ガイドラインの [入力チェック](#) も合わせて参照すること。以下に、主な比較について示す。

#### 主な比較一覧

比較対象	TERASOLUNA Server 5.x	TERASOLUNA Batch 5.x
使用できる入力チェックルール	TERASOLUNA Server 5.xと同様	
ルールを付与する対象	フォームクラス	DTO
チェックの実行方法	Controllerに@Validatedアノテーションを付与する	ValidatorクラスのAPIをコールする
エラーメッセージの設定	TERASOLUNA Server 5.x 開発ガイドラインの <a href="#">エラーメッセージの定義</a> と同様	
エラーメッセージの出力先	画面	ログ等

なお、本節で説明対象とする入力チェックは、主にステップが処理する入力データを対象とする。ジョブパラメータのチェックについては[パラメータの妥当性検証](#)を参照のこと。

#### 6.1.1.1. 入力チェックの分類

入力チェックは、単項目チェック、相関項目チェックに分類される。

#### 設定内容の項目一覧

種類	説明	例	実現方法
単項目チェック	単一のフィールドで完結するチェック	入力必須チェック 桁チェック 型チェック	Bean Validation(実装ライブラリとしてHibernate Validatorを使用)

種類	説明	例	実現方法
相関項目チェック	複数のフィールドを比較するチェック	数値の大小比較 日付の前後比較	<code>org.springframework.validation.Validator</code> インターフェースを実装したValidationクラス または Bean Validation

Springは、Java標準であるBean Validationをサポートしている。単項目チェックには、このBean Validationを利用する。相関項目チェックの場合は、Bean ValidationまたはSpringが提供している`org.springframework.validation.Validator`インターフェースを利用する。

この点は、TERASOLUNA Server 5.x 開発ガイドラインの [入力チェックの分類](#) と同様である。

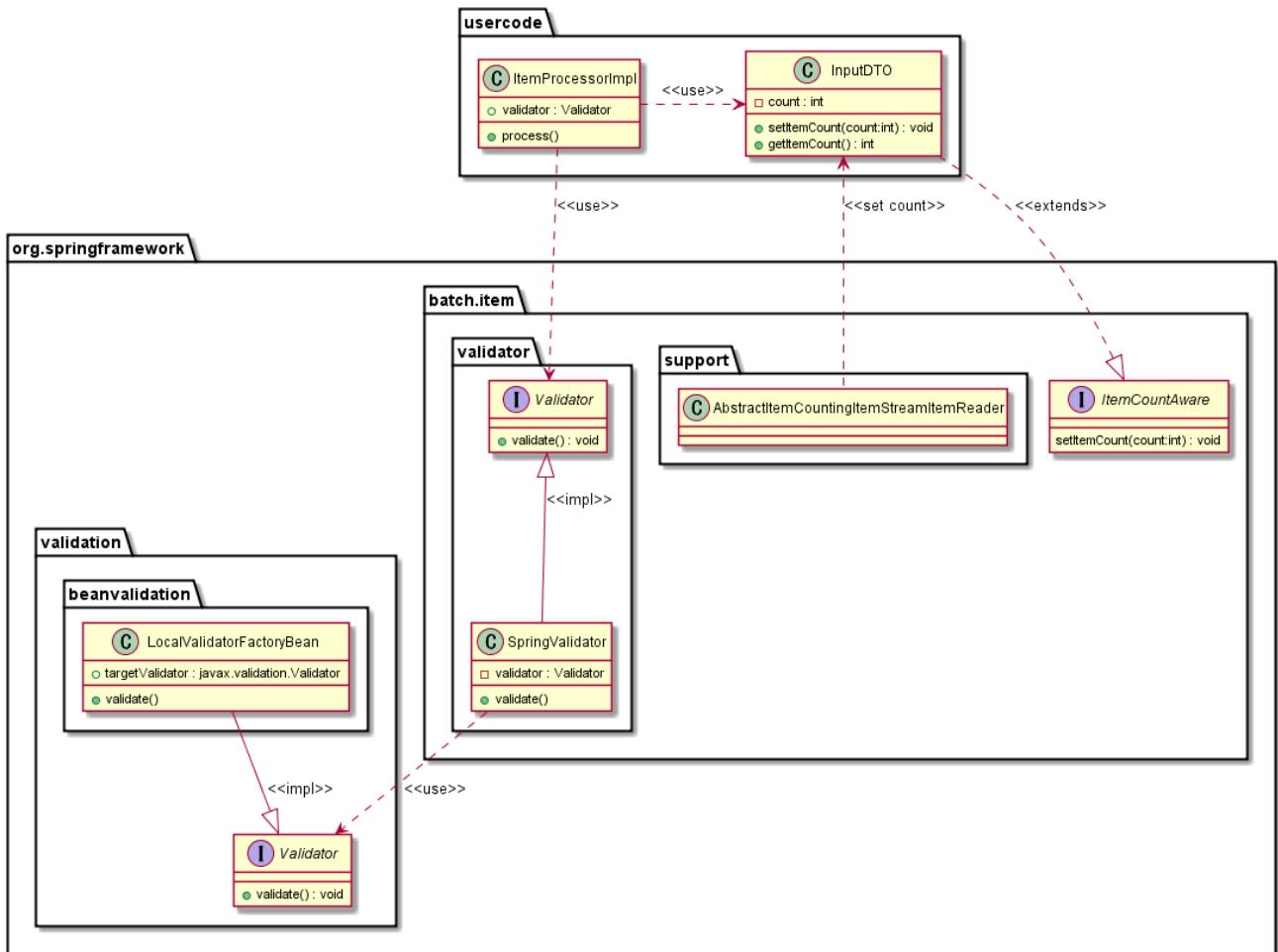
#### 6.1.1.2. 入力チェックの全体像

チャンクモデル、タスクレットモデルにて入力チェックを行うタイミングは以下のとおりである。

- チャンクモデルの場合は`ItemProcessor`で行う。
- タスクレットモデルの場合は`Tasklet#execute()`にて、任意のタイミングで行う。

チャンクモデル、タスクレットモデルにおいて入力チェックの実装方法は同様となるため、ここではチャンクモデルの`ItemProcessor`で入力チェックを行う場合について説明する。

まず、入力チェックの全体像を説明する。入力チェックに関連するクラスの関係は以下のとおりである。



## 入力チェックの関連クラス

- `ItemProcessor`に、`org.springframework.batch.item.validator.Validator`の実装である`org.springframework.batch.item.validator.SpringValidator`をインジェクションしvalidateメソッドを実行する。
  - `SpringValidator`は内部に`org.springframework.validation.Validator`を保持し、validateメソッドを実行する。  
いわば、`org.springframework.validation.Validator`のラッパーといえる。  
`org.springframework.validation.Validator`の実装は、  
`org.springframework.validation.beanvalidation.LocalValidatorFactoryBean`となる。このクラスを通じてHibernate Validatorを使用する。
- 何件目のデータで入力チェックエラーになったのかを判別するため  
に`org.springframework.batch.item.ItemCountAware`を入力DTOに実装する。

データ件数の設定



`ItemCountAware#setItemCount`は`AbstractItemCountingItemStreamItemReader`によって設定される。よって、タスクレットモデルで`ItemReader`を使わない場合、更新されない。この場合は何件目のデータでエラーになったかはユーザにて設定すること。

`javax.validation.Validator`や`org.springframework.validation.Validator`といったバリデータは直接使用しない。

`javax.validation.Validator`や`org.springframework.validation.Validator`といったバリデータは直接使用せず、`org.springframework.batch.item.validator.SpringValidator`を使用する。



`SpringValidator`は`org.springframework.validation.Validator`のラッパーである。`SpringValidator`は発生した例外を`BindException`にラップし、`ValidationException`としてスローする。そのため、`ValidationException`を通して`BindException`にアクセスでき、柔軟なハンドリングがしやすくなる。

一方、`javax.validation.Validator`や`org.springframework.validation.Validator`といったバリデータを直接使用すると、バリデーションエラーになった情報を処理する際に煩雑なロジックになってしまふ。

`org.springframework.batch.item.validator.ValidatingItemProcessor`は使用しない`org.springframework.validation.Validator`による入力チェックは、Spring Batchが提供する`ValidatingItemProcessor`を使用しても実現可能である。

しかし、以下の理由により状況によっては拡張を必要としてしまうため、実装方法を統一する観点より使用しないこととする。



- 入力チェックエラーをハンドリングし処理を継続することができない。
- 入力チェックエラーとなったデータに対して柔軟な対応を行うことができない。
  - 入力チェックエラーとなったデータに対しての処理は、利用者によって多種多様(ログ出力のみ、エラーデータを別ファイルに退避する、など)となると想定される。

### 6.1.2. How to use

先にも述べたが、入力チェックの実現方法は以下のとおりTERASOLUNA Server 5.xと同様である。

- 単項目チェックは、Bean Validationを利用する。
- 相関項目チェックは、Bean ValidationまたはSpringが提供している`org.springframework.validation.Validator`インターフェースを利用する。

入力チェックの方法について以下の順序で説明する。

- 各種設定
- 入力チェックルールの定義
- 入力チェックの実施
- 入力チェックエラーのハンドリング

### 6.1.2.1. 各種設定

入力チェックにはHibernate Validatorを使用する。ライブラリの依存関係にHibernate Validatorの定義があり、必要なBean定義が存在することを確認する。これらは、TERASOLUNA Batch 5.xが提供するブランクプロジェクトではすでに設定済である。

依存ライブラリの設定例

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
</dependency>
```

*launch-context.xml*

```
<bean id="validator" class="org.springframework.batch.item.validator.SpringValidator"
    p:validator-ref="beanValidator"/>

<bean id="beanValidator"
    class="org.springframework.validation.beanvalidation.LocalValidatorFactoryBean"
/>
```

エラーメッセージの設定

先にも述べたが、エラーメッセージの設定については、TERASOLUNA Server 5.x 開発ガイドラインの [エラーメッセージの定義](#) を参照すること。

### 6.1.2.2. 入力チェックルールの定義

入力チェックのルールを実装する対象はItemReaderを通じて取得するDTOである。ItemReaderを通じて取得するDTOは以下の要領で実装する。

- 何件目のデータで入力チェックエラーになったのかを判別するため  
に、org.springframework.batch.item.ItemCountAwareを実装する。
  - setItemCountメソッドにて引数で受けた現在処理中のitemが読み込み何件目であるかをあらわす  
数値をクラスフィールドに保持する。
- 入力チェックルールを定義する。
  - TERASOLUNA Server 5.x 開発ガイドラインの [入力チェック](#) を参照。

以下に、入力チェックルールを定義したDTOの例を示す。

## 入力チェックルールを定義したDTOの例

```
public class VerificationSalesPlanDetail implements ItemCountAware { // (1)

    private int count;

    @NotEmpty
    @Size(min = 1, max = 6)
    private String branchId;

    @NotNull
    @Min(1)
    @Max(9999)
    private int year;

    @NotNull
    @Min(1)
    @Max(12)
    private int month;

    @NotEmpty
    @Size(min = 1, max = 10)
    private String customerId;

    @NotNull
    @DecimalMin("0")
    @DecimalMax("9999999999")
    private BigDecimal amount;

    @Override
    public void setItemCount(int count) {
        this.count = count; // (2)
    }

    // omitted getter/setter
}
```

### 設定内容の項目一覧

項目番	説明
(1)	ItemCountAwareクラスを実装し、setItemCountメソッドをオーバーライドする。 ItemCountAware#setItemCount()は、ItemReaderが読み込んだデータが何件目であるかを引数に渡される。
(2)	引数で受けるcountをクラスフィールドに保持する。 この値は、何件目のデータで入力チェックエラーになったのかを判別するため使用する。

### 6.1.2.3. 入力チェックの実施

入力チェックの実施方法について説明する。入力チェック実施は以下の要領で実装する。

- `ItemProcessor`の実装にて、`org.springframework.batch.item.validator.Validator#validate()`を実行する。
  - `Validator`には`SpringValidator`のインスタンスをインジェクトして使用する。
- 入力チェックエラーをハンドリングする。詳細は[入力チェックエラーのハンドリング](#)を参照すること。

入力チェックの実施例を以下に示す。

入力チェックを実施する例

```
@Component
public class ValidateAndContinueItemProcessor implements ItemProcessor<VerificationSalesPlanDetail, SalesPlanDetail> {
    @Inject // (1)
    Validator<VerificationSalesPlanDetail> validator;

    @Override
    public SalesPlanDetail process(VerificationSalesPlanDetail item) throws Exception
    {
        try { // (2)
            validator.validate(item); // (3)
        } catch (ValidationException e) {
            // omitted exception handling
        }

        SalesPlanDetail salesPlanDetail = new SalesPlanDetail();
        // omitted business logic

        return salesPlanDetail;
    }
}
```

設定内容の項目一覧

項目番号	説明
(1)	<code>SpringValidator</code> のインスタンスをインジェクトする。 <code>org.springframework.batch.item.validator.Validator</code> の型引数には、 <code>ItemReader</code> を通じて取得するDTOを設定する。
(2)	入力チェックエラーをハンドリングする。 例では例外をtry/catchで捕捉する方法で処理している。 詳細は <a href="#">入力チェックエラーのハンドリング</a> を参照すること。
(3)	<code>ItemReader</code> を通じて取得するDTOを引数として <code>Validator#validate()</code> を実行する。

#### 6.1.2.4. 入力チェックエラーのハンドリング

入力チェックエラーが発生した場合の選択肢は以下の2択となる。

1. 入力チェックエラーが発生した時点で処理を打ち切り、ジョブを異常終了させる。

2. 入力チェックエラーが発生したことをログ等に残し、後続データの処理は継続する。その後、ジョブ終了時に、ジョブを警告終了させる。

#### 6.1.2.4.1. 処理を異常終了する場合

例外発生時に処理を異常終了するためには、`java.lang.RuntimeException`またはそのサブクラスをスローする。

例外発生時にログ出力等の処理を行う方法は以下の2つとおりがある。

1. 例外をtry/catchで捕捉し、例外をスローする前に行う。
2. 例外をtry/catchで捕捉せず、`ItemProcessListener`を実装し`onProcessError`メソッドにて行う。
  - `ItemProcessListener#onProcessError()`は`@OnProcessError`アノテーションを使用して実装してもよい。 詳細は、[リスナー](#)を参照のこと。

例外発生時に、例外情報をログ出力し、処理を異常終了する例を以下に示す。

## try/catchによるハンドリング例

```
@Component
public class ValidateAndAbortItemProcessor implements ItemProcessor<VerificationSalesPlanDetail, SalesPlanDetail> {
    /**
     * Logger.
     */
    private static final Logger logger = LoggerFactory.getLogger(ValidateAndAbortItemProcessor.class);

    @Inject
    Validator<VerificationSalesPlanDetail> validator;

    @Override
    public SalesPlanDetail process(VerificationSalesPlanDetail item) throws Exception
    {
        try { // (1)
            validator.validate(item); // (2)
        } catch (ValidationException e) {
            // (3)
            logger.error("Exception occurred in input validation at the {} th item.
[message:{}]",
                         item.getCount(), e.getMessage());
            throw e; // (4)
        }

        SalesPlanDetail salesPlanDetail = new SalesPlanDetail();
        // omitted business logic

        return salesPlanDetail;
    }
}
```

## 設定内容の項目一覧

項目番	説明
(1)	try/catchにて例外を捕捉する。
(2)	入力チェックを実行する。
(3)	例外をスローする前にログ出力処理を行う。
(4)	例外をスローする。 <code>org.springframework.batch.item.validator.ValidationException</code> は <code>RuntimeException</code> のサブクラスであるため、そのままスローしなおしてよい。

## ItemProcessListener#OnProcessErrorによるハンドリング例

```
@Component
public class ValidateAndAbortItemProcessor implements ItemProcessor<VerificationSalesPlanDetail, SalesPlanDetail> {

    /**
     * Logger.
     */
    private static final Logger logger = LoggerFactory.getLogger(ValidateAndAbortItemProcessor.class);

    @Inject
    Validator<VerificationSalesPlanDetail> validator;

    @Override
    public SalesPlanDetail process(VerificationSalesPlanDetail item) throws Exception
    {
        validator.validate(item); // (1)

        SalesPlanDetail salesPlanDetail = new SalesPlanDetail();
        // omitted business logic

        return salesPlanDetail;
    }

    @OnProcessError // (2)
    void onProcessError(VerificationSalesPlanDetail item, Exception e) {
        // (3)
        logger.error("Exception occurred in input validation at the {} th item.
[message:{}]", item.getCount() ,e.getMessage());
    }
}
```

### 設定内容の項目一覧

項目番	説明
(1)	入力チェックを実行する。
(2)	ItemProcessListener#onProcessError()を@OnProcessErrorアノテーションを使用して実装する。
(3)	例外をスローする前にログ出力処理を行う。

#### *ItemProcessListener#onProcessError()*使用時の注意点

*onProcessError*メソッドの利用は業務処理と例外ハンドリングを切り離すことができるためソースコードの可読性、保守性の向上等に有用である。

しかし、上記の例でハンドリング処理を行っている*ValidationException*以外の例外が発生した場合も同じメソッドが実行されるため注意が必要である。



*ItemProcessor#process()*におけるログ出力を例外によって出力し分ける場合は、*onProcessError*メソッドにて発生した例外の種類を判定して例外処理を行う必要がある。これが煩雑である場合は、try/catchによるハンドリングにて入力チェックエラーのみを処理し、それ以外はリスナーに移譲するように責務を分担するとよい。

#### 6.1.2.4.2. エラーレコードをスキップする場合

入力チェックエラーが発生したレコードの情報をログ出力等を行った後、エラーが発生したレコードをスキップして後続データの処理を継続する場合は以下の要領で実装する。

- 例外をtry/catchで捕捉する。
- 例外発生時のログ出力等を行う。
- ItemProcessor#process()*の返り値としてnullを返却する。
  - nullを返却することで入力チェックエラーが発生したレコードは後続の処理対象(*ItemWriter*による出力)に含まれなくなる。

## ItemProcessorによるスキップ例

```
@Component
public class ValidateAndContinueItemProcessor implements ItemProcessor<VerificationSalesPlanDetail, SalesPlanDetail> {
    /**
     * Logger.
     */
    private static final Logger logger = LoggerFactory.getLogger(ValidateAndContinueItemProcessor.class);

    @Inject
    Validator<VerificationSalesPlanDetail> validator;

    @Override
    public SalesPlanDetail process(VerificationSalesPlanDetail item) throws Exception
    {
        try { // (1)
            validator.validate(item); // (2)
        } catch (ValidationException e) {
            // (3)
            logger.warn("Skipping item because exception occurred in input validation
at the {} th item. [message:{}]",
                item.getCount(), e.getMessage());
        } // (4)
        return null; // skipping item
    }

    SalesPlanDetail salesPlanDetail = new SalesPlanDetail();
    // omitted business logic

    return salesPlanDetail;
    }
}
```

### 設定内容の項目一覧

項目番	説明
(1)	try/catchにて例外を捕捉する。
(2)	入力チェックを実行する。
(3)	nullを返却する前にログ出力処理を行う。
(4)	nullを返却することで当該データをスキップし次のデータ処理へ移る。

#### 6.1.2.4.3. 終了コードの設定

入力チェックエラーが発生した場合、入力チェックエラーが発生しなかった場合とジョブの状態を区別するためには必ず正常終了ではない終了コードを設定すること。

入力チェックエラーが発生したデータをスキップした場合、異常終了した場合においても終了コードの設

定は必須である。

終了コードの設定方法については、[ジョブの管理](#)を参照すること。

## 6.2. 例外ハンドリング

### 6.2.1. Overview

ジョブ実行時に発生する例外のハンドリング方法について説明する。

本機能は、チャンクモデルとタスクレットモデルとで使い方が異なるため、それぞれについて説明する。

まず、例外の分類について説明し、例外の種類に応じたハンドリング方法を説明する。

#### 6.2.1.1. 例外の分類

ジョブ実行時に発生する例外は、以下の3つに分類される。

#### 例外の分類一覧

項目番号	分類	説明	例外の種類
(1)	ジョブの再実行(パラメータ、入力データの変更/修正など)によって発生原因が解消できる例外	ジョブの再実行で発生原因が解消できる例外は、アプリケーションコードで例外をハンドリングし、例外処理を行う。	ビジネス例外 正常稼働時に発生するライブリ例外
(2)	ジョブの再実行によって発生原因が解消できない例外	ジョブの再実行で発生原因が解消できる例外は、以下のパターンにてハンドリングする。 1. <a href="#">StepListener</a> で例外の捕捉が可能な場合は、 アプリケーションコードで例外をハンドリングする。 2. <a href="#">StepListener</a> で例外の捕捉が不可能な場合は、 フレームワークで例外処理をハンドリングする。	システム例外 予期しないシステム例外 致命的なエラー
(3)	(非同期実行時に)ジョブ要求のリクエスト不正により発生する例外	ジョブ要求のリクエスト不正により発生する例外は、フレームワークで例外処理をハンドリングし、例外処理を行う。 ▲ <a href="#">非同期実行(DBポーリング)</a> の場合は、 ポーリング処理ではジョブ要求に対する妥当性検証をしない。そのため、ジョブ要求を登録するアプリケーションで事前にリクエストに対する入力チェックが行われていることが望ましい。 ▲ <a href="#">非同期実行(Webコンテナ)</a> の場合は、 Webアプリケーションにより事前にリクエストに対する入力チェックが行われていることを前提としている。 そのため、ジョブ要求やリクエストを受け付けるアプリケーションで例外ハンドリングを行う。	ジョブ要求リクエスト不正エラー



例外処理内でトランザクショナルな処理は避ける

例外処理内でデータベースへの書き込みを始めとするトランザクショナルな処理を行うと、二次例外を引き起こしてしまう可能性がある。例外処理は、解析用ログ出力と終了コード設定を基本とすること。

### 6.2.1.2. 例外の種類

例外の種類について説明する。

#### 6.2.1.2.1. ビジネス例外

ビジネス例外とは、ビジネスルールの違反を検知したことを通知する例外である。

本例外は、ステップのロジック内で発生させる。

アプリケーションとして想定される状態なので、システム運用者による対処は不要である。

ビジネス例外の例

- 在庫引当時に在庫切れの場合
- 予定日より日数が超過した場合
- etc ...

該当する例外クラス



- `java.lang.RuntimeException`またはそのサブクラス
  - ビジネス例外クラスをユーザにて作成することを推奨する

#### 6.2.1.2.2. 正常稼働時に発生するライブラリ例外

正常稼働時に発生するライブラリ例外とは、フレームワーク、およびライブラリ内で発生する例外のうち、システムが正常稼働している時に発生する可能性のある例外のことを指す。

フレームワーク、およびライブラリ内で発生する例外とは、Spring Frameworkや、その他のライブラリ内で発生する例外クラスを対象とする。

アプリケーションとして想定される状態なので、システム運用者による対処は不要である。

正常稼働時に発生するライブラリ例外の例

- オンライン処理との`排他制御`で発生する楽観ロック例外
- 複数ジョブやオンライン処理からの同一データを同時登録する際に発生する一意制約例外
- etc ...

該当する例外クラス



- `org.springframework.dao.EmptyResultDataAccessException` (楽観ロックをした時、データ更新件数が0件の場合に発生する例外)
- `org.springframework.dao.DuplicateKeyException` (一意制約違反となった場合に発生する例外)
- etc ...

#### 6.2.1.2.3. システム例外

システム例外とは、システムが正常稼働している時に、発生してはいけない状態を検知したことを見つける例外である。

本例外は、ステップのロジック内で発生させる。

システム運用者による対処が必要となる。

システム例外の例

- 事前に存在しているはずのマスタデータ、ディレクトリ、ファイルなどが存在しない場合。
- フレームワーク、ライブラリ内で発生する検査例外のうち、システム異常に分類される例外を捕捉した場合(ファイル操作時のIOExceptionなど)。
- etc...

該当する例外クラス



- `java.lang.RuntimeException`またはそのサブクラス
  - システム例外クラスを作成することを推奨する

#### 6.2.1.2.4. 予期しないシステム例外

予期しないシステム例外とは、システムが正常稼働している時には発生しない非検査例外である。

システム運用者による対処、またはシステム開発者による解析が必要となる。

予期しないシステム例外は、以下の処理をする以外はハンドリングしない。ハンドリングした場合は、例外を再度スローすること。

- 捕捉例外を解析用にログ出力を行い、該当する終了コードの設定する。

予期しないシステム例外の例

- アプリケーション、フレームワーク、ライブラリにバグが潜んでいる場合。
- DBサーバなどがダウンしている場合。
- etc...

該当する例外クラス



- `java.lang.NullPointerException`(バグ起因で発生する例外)
- `org.springframework.dao.DataAccessResourceFailureException`(DBサーバがダウンしている場合に発生する例外)
- etc ...

#### 6.2.1.2.5. 致命的なエラー

致命的なエラーとは、システム(アプリケーション)全体に影響を及ぼす、致命的な問題が発生している事を通知するエラーである。

システム運用者、またはシステム開発者による対処・リカバリが必要となる。

致命的なエラーは、以下の処理をする以外はハンドリングしない。ハンドリングした場合は、例外を再度スローすること。

- ・捕捉例外を解析用にログ出を行い、該当する終了コードの設定する。

#### 致命的なエラーの例

- ・Java仮想マシンで使用できるメモリが不足している場合。
- ・etc...

##### 該当する例外クラス



- ・`java.lang.Error`を継承しているクラス
  - ・`java.lang.OutOfMemoryError` (メモリ不足時に発生するエラー)など
  - ・etc ...

#### 6.2.1.2.6. ジョブ要求リクエスト不正エラー

ジョブ要求リクエスト不正エラーとは、非同期実行時にジョブ要求のリクエストに問題が発生していることを通知するエラーである。

システム運用者による対処・リカバリが必要となる。

ジョブ要求リクエスト不正エラーは、ジョブ要求のリクエストを処理するアプリケーションでの例外ハンドリングを前提にするため、本ガイドラインでは説明はしない。

#### 6.2.1.3. 例外への対応方法

例外への対応方法について説明する。

例外への対応パターンは次のとおり。

1. 例外発生時にジョブの継続可否を決める(3種類)
2. 中断したジョブの再実行方法を決める(2種類)

##### ジョブの継続可否を決定する方法

項目番号	例外への対応方法	説明
(1)	スキップ	エラーレコードをスキップし、処理を継続する。
(2)	リトライ	エラーレコードを指定した条件(回数、時間等)に達するまで再処理する。
(3)	処理中断	処理を中断する。



例外が発生していないくとも、ジョブが想定以上の処理時間になったため処理途中で停止する場合がある。  
この場合は、[ジョブの停止](#)を参照。

##### 中断したジョブの再実行方法

項目番号	例外への対応方法	説明
(1)	ジョブのリラン	中断したジョブを最初から再実行する。
(2)	ジョブのリストート	中断したジョブを中断した箇所から再実行する。

中断したジョブの再実行方法についての詳細は、[処理の再実行](#)を参照してほしい。

#### 6.2.1.3.1. スキップ

スキップとは、バッチ処理を止めずにエラーデータを飛ばして処理を継続する方法である。

スキップを行う例

- ・入力データ内に不正なレコードが存在する場合
- ・ビジネス例外が発生した場合
- ・etc ...

##### スキップレコードの再処理



スキップを行う場合は、スキップした不正なレコードについてどのように対応するか設計すること。不正なレコードを抽出して再処理する場合、次回実行時に含めて処理する場合、などといった方法が考えられる。

#### 6.2.1.3.2. リトライ

リトライとは、特定の処理に失敗したレコードに対して指定した回数や時間に達するまで再試行を繰り返す対応方法である。

処理失敗の原因が実行環境に依存しており、かつ、時間の経過により解決される見込みのある場合にのみ用いる。

リトライを行う例

- ・排他制御により、処理対象のレコードがロックされている場合
- ・ネットワークの瞬断によりメッセージ送信が失敗する場合
- ・etc ...

##### リトライの適用



リトライをあらゆる場面で適用してしまうと、異常発生時に処理時間がむやみに伸びてしまい、異常の検出が遅れる危険がある。

よって、リトライは処理のごく一部に適用することが望ましく、その対象は外部システム連携など信頼性が担保しにくいものに限定するとよい。

#### 6.2.1.3.3. 処理中断

処理中断とは、文字どおり処理を途中で中断する対応方式である。

処理の継続が不可能な内容のエラーが検知された場合や、レコードのスキップを許容しない要件の場合に用いる。

処理中断を行う例

- ・入力データ内に不正なレコードが存在する場合
- ・ビジネス例外が発生した場合
- ・etc ...

## 6.2.2. How to use

例外ハンドリングの実現方法について説明をする。

バッチアプリケーション運用時のユーザインターフェースはログが主体である。よって、例外発生の監視もログを通じて行うことになる。

Spring Batch では、ステップ実行時に例外が発生した場合はログを出力し異常終了するため、ユーザにて追加実装をせずとも要件を満たせる可能性がある。以降の説明は、ユーザにてシステムに応じたログ出力をを行う必要があるときのみ、ピンポイントに実装するとよい。すべての処理を実装しなくてはならないケースは基本的にはない。

例外ハンドリングの共通であるログ設定については、[ロギング](#)を参照。

### 6.2.2.1. ステップ単位の例外ハンドリング

ステップ単位での例外ハンドリング方法について説明する。

#### *ChunkListener*インターフェースによる例外ハンドリング

処理モデルによらず、発生した例外を統一的にハンドリングしたい場合は、[ChunkListener](#)インターフェースを利用する。

チャンクよりスコープの広い、ステップやジョブのリスナーを利用して実現できるが、出来る限り発生した直後にハンドリングすることを重視し、[ChunkListener](#)を採用する。

各処理モデルごとの例外ハンドリング方法は以下のとおり。

#### チャンクモデルにおける例外ハンドリング

Spring Batch 提供の各種 Listener インターフェースを使用して機能を実現する。

#### タスクレットモデルにおける例外ハンドリング

タスクレット実装内にて独自に例外ハンドリングを実装する。

*ChunkListener*で統一的にハンドリングできるのはなぜか

[ChunkListener](#)によってタスクレット実装内で発生した例外をハンドリングできることに違和感を感じるかもしれない。これは、Spring Batchにおいてビジネスロジックの実行はチャンクを基準に考えられており、1回のタスクレット実行は、1つのチャンク処理として扱われているためである。



この点は[org.springframework.batch.core.step.tasklet.Tasklet](#)のインターフェースにも表れている。

```
public interface Tasklet {  
    RepeatStatus execute(StepContribution contribution,  
                         ChunkContext chunkContext) throws Exception;  
}
```

### 6.2.2.1.1. ChunkListenerインターフェースによる例外ハンドリング

ChunkListenerインターフェースの`afterChunkError`メソッドを実装する。

`afterChunkError`メソッドの引数である`ChunkContext`から`ChunkListener.ROLLBACK_EXCEPTION_KEY`をキーにしてエラー情報を取得する。

リスナーの設定方法については、[リスナーの設定](#)を参照。

*ChunkListener*の実装例

```
@Component
public class ChunkAroundListener implements ChunkListener {

    private static final Logger logger =
        LoggerFactory.getLogger(ChunkAroundListener.class);

    @Override
    public void beforeChunk(ChunkContext context) {
        logger.info("before chunk. [context:{}]", context);
    }

    @Override
    public void afterChunk(ChunkContext context) {
        logger.info("after chunk. [context:{}]", context);
    }

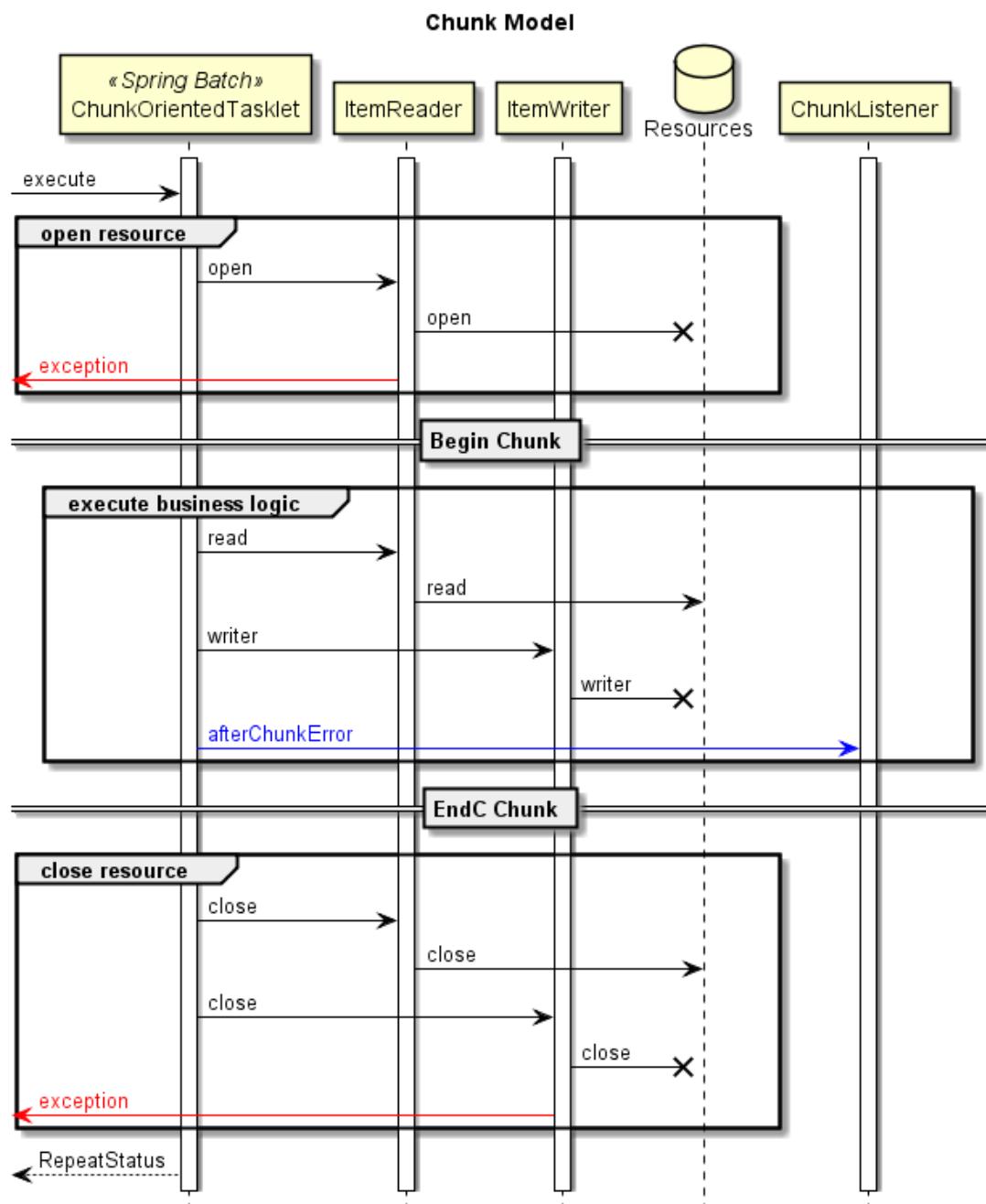
    // (1)
    @Override
    public void afterChunkError(ChunkContext context) {
        logger.error("Exception occurred while chunk. [context:{}]", context,
            context.getAttribute(ChunkListener.ROLLBACK_EXCEPTION_KEY)); // (2)
    }
}
```

説明

項番	説明
(1)	<code>afterChunkError</code> メソッドを実装する。
(2)	<code>ChunkContext</code> から <code>ChunkListener.ROLLBACK_EXCEPTION_KEY</code> をキーにしてエラー情報を取得する。 この例では、取得した例外のスタックトレースをログ出力している。

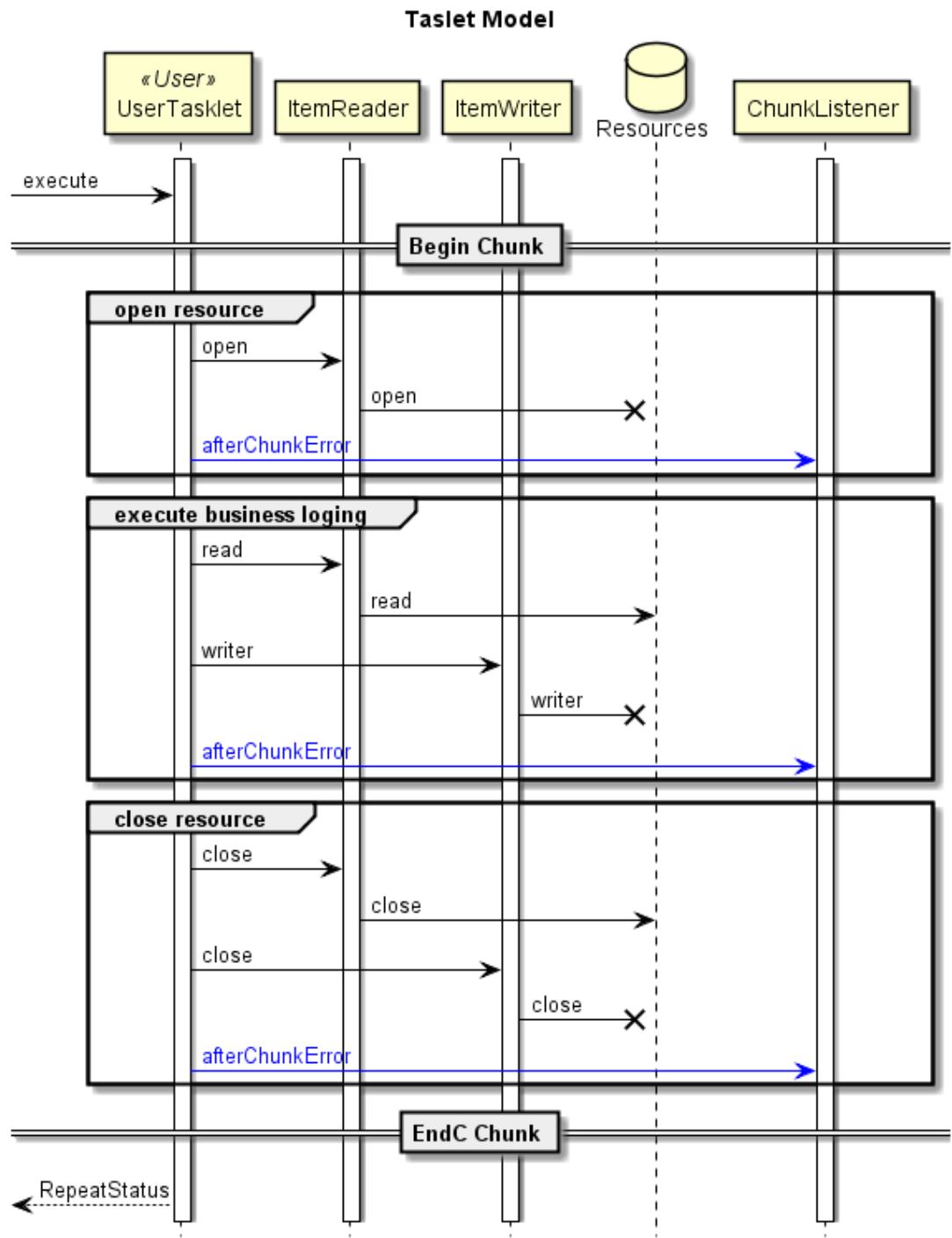
処理モデルの違いによる*ChunkListener*の挙動の違い

チャunkモデルでは、リソースのオープン・クローズで発生した例外は、*ChunkListener*インターフェースが捕捉するスコープ外となる。そのため、`afterChunkError`メソッドでハンドリングが行われない。概略図を以下に示す。



チャンクモデルでの例外ハンドリング概略図

タスクレットモデルでは、リソースのオープン・クローズで発生した例外は、ChunkListenerインターフェースが捕捉するスコープ内となる。そのため、**afterChunkError**メソッドでハンドリングが行われる。概略図を以下に示す。



タスクレットモデルでの例外ハンドリング概略図

この挙動の差を吸収して統一的に例外をハンドリングしたい場合は、[StepExecutionListener](#)インターフェースで例外の発生有無をチェックすることで実現できる。ただし、[ChunkListener](#)よりも実装が少々複雑になる。

### StepExecutionListenerの実装例

```
@Component
public class StepErrorLoggingListener implements StepExecutionListener {
    private static final Logger logger =
        LoggerFactory.getLogger(StepErrorLoggingListener.class);

    @Override
    public void beforeStep(StepExecution stepExecution) {
        // do nothing.
    }

    // (1)
    @Override
    public ExitStatus afterStep(StepExecution stepExecution) {
        // (2)
        List<Throwable> exceptions = stepExecution
            .getFailureExceptions();
        // (3)
        if (exceptions.isEmpty()) {
            return ExitStatus.COMPLETED;
        }

        // (4)
        logger.info("This step has occurred some exceptions as follow.
        " +
                    "[step-name:{}] [size:{}]",
                    stepExecution.getStepName(), exceptions.size());
        exceptions.forEach(th -> logger.error(
            "exception has occurred in job.", th));
        return ExitStatus.FAILED;
    }
}
```

#### 説明

項目番号	説明
(1)	afterStepメソッドを実装する。
(2)	引数のstepExecutionからエラー情報を取得する。複数の例外をまとめて扱う必要がある点に注意する。
(3)	エラー情報がない場合は、正常終了とする。
(4)	エラー情報がある場合は、例外ハンドリングを行う。 この例では、発生した例外をすべてスタックトレース付きのログ出力を行っている。

#### 6.2.2.1.2. チャンクモデルにおける例外ハンドリング

チャンクモデルでは、 StepListenerを継承したListenerで例外ハンドリングする。

リスナーの設定方法については、[リスナーの設定](#)を参照。

### コーディングポイント(ItemReader編)

[ItemReadListener](#)インターフェースの `onReadError` メソッドを実装することで、ItemReader内で発生した例外をハンドリングする。

*ItemReadListener#onReadError*の実装例

```
@Component
public class CommonItemReadListener implements ItemReadListener<Object> {

    private static final Logger logger =
        LoggerFactory.getLogger(CommonItemReadListener.class);

    // omitted.

    // (1)
    @Override
    public void onReadError(Exception ex) {
        logger.error("Exception occurred while reading.", ex); // (2)
    }

    // omitted.
}
```

#### 説明

項番	説明
(1)	<code>onReadError</code> メソッドを実装する。
(2)	例外ハンドリングを実装する この例では、引数から取得した例外のスタックトレースをログ出力している。

### コーディングポイント(ItemProcessor編)

ItemProcessorでの例外ハンドリングには、2つの方法があり、要件に応じて使い分ける。

1. ItemProcessor 内でtry～catchをする方法
2. [ItemProcessListener](#)インターフェースを使用する方法

使い分ける理由について説明する。

ItemProcessorの処理内で例外発生時に実行される `onProcessError` メソッドの引数は、処理対処のアイテムと例外の2つである。

システムの要件によっては、[ItemProcessListener](#)インターフェース内でログ出力等の例外をハンドリングする際に、この2つの引数で要件を満たせない場合が出てくる。その場合は、ItemProcessor内でtry～catchにて例外をcatchし例外ハンドリング処理を行うことを推奨する。

注意点として、ItemProcessor内でtry～catchを実装した上で、[ItemProcessListener](#)インターフェースを実装すると二重処理になる場合があるため、注意が必要である。

きめ細かい例外ハンドリングを行いたい場合は、ItemProcessor内でtry～catchをする方法を採用するこ

ヒ。

それぞれの方法について説明する。

### ItemProcessor 内でtry～catchする方法

きめ細かい例外ハンドリングが必要になる場合はこちらを使用する。

後述するスキップの項で説明するが、エラーレコードの[スキップ](#)を行う際にはこちらを使用することとなる。

### ItemProcessor内でtry～catchする実装例

```
@Component
public class AmountCheckProcessor implements
    ItemProcessor<SalesPerformanceDetail, SalesPerformanceDetail> {

    // omitted.

    @Override
    public SalesPerformanceDetail process(SalesPerformanceDetail item)
        throws Exception {
        // (1)
        try {
            checkAmount(item.getAmount(), amountLimit);
        } catch (ArithmetricException ae) {
            // (2)
            logger.error(
                "Exception occurred while processing. [item:{}]", item, ae);
            // (3)
            throw new IllegalStateException("check error at processor.", ae);
        }
        return item;
    }
}
```

### 説明

項目番号	説明
(1)	try～catchで実装する。ここでは、特定の例外(ArithmetricException)のみ特別なハンドリングをしている。
(2)	例外ハンドリングを実装する この例では、引数から取得した例外のスタックトレースをログ出力している。
(3)	トランザクションのロールバック例外をスローする。 また、この例外スローによりItemProcessListenerで共通の例外ハンドリングをすることもできる。

### ItemProcessListenerインターフェースを使用する方法

業務例外に対するハンドリングが共通化できる場合はこちらを使用する。

## ItemProcessListener#onProcessErrorの実装例

```
@Component
public class CommonItemProcessListener implements ItemProcessListener<Object, Object>
{
    private static final Logger logger =
        LoggerFactory.getLogger(CommonItemProcessListener.class);

    // omitted.

    // (1)
    @Override
    public void onProcessError(Object item, Exception e) {
        // (2)
        logger.error("Exception occurred while processing. [item:{}]", item, e);
    }

    // omitted.
}
```

### 説明

項目番	説明
(1)	onProcessErrorメソッドを実装する。
(2)	例外ハンドリングを実装する この例では、引数から取得した処理対象データと例外のスタックトレースをログ出力している。

### コーディングポイント(ItemWriter編)

ItemClickListenerインターフェースの onWriteErrorメソッドを実装することで、ItemWriter内で発生した例外をハンドリングする。

## ItemClickListener#onWriteErrorの実装例

```
@Component
public class CommonItemClickListener implements ItemWriteListener<Object> {

    private static final Logger logger =
        LoggerFactory.getLogger(CommonItemClickListener.class);

    // omitted.

    // (1)
    @Override
    public void onWriteError(Exception ex, List item) {
        // (2)
        logger.error("Exception occurred while processing. [items:{}]", item, ex);
    }

    // omitted.
}
```

### 説明

項番	説明
(1)	onWriteErrorメソッドを実装する。
(2)	例外ハンドリングを実装する この例では、引数から取得した出力対象のチャunkと例外のスタックトレースをログ出力している。

#### 6.2.2.1.3. タスクレットモデルにおける例外ハンドリング

タスクレットモデルの例外ハンドリングはタスクレット内で独自に実装する。

トランザクション処理を行う場合は、ロールバックさせるために必ず例外を再度スローすること。

## タスクレットモデルでの例外ハンドリング実装例

```
@Component
public class SalesPerformanceTasklet implements Tasklet {

    private static final Logger logger =
        LoggerFactory.getLogger(SalesPerformanceTasklet.class);

    // omitted

    @Override
    public RepeatStatus execute(StepContribution contribution,
                               ChunkContext chunkContext) throws Exception {

        // (1)
        try {
            reader.open(chunkContext.getStepContext().getStepExecution()
                .getExecutionContext());

            List<SalesPerformanceDetail> items = new ArrayList<>(10);
            SalesPerformanceDetail item = null;
            do {
                // Pseudo operation of ItemReader
                // omitted

                // Pseudo operation of ItemProcessor
                checkAmount(item.getAmount(), amountLimit);

                // Pseudo operation of ItemWriter
                // omitted

            } while (item != null);
        } catch (Exception e) {
            logger.error("exception in tasklet.", e); // (2)
            throw e; // (3)
        } finally {
            try {
                reader.close();
            } catch (Exception e) {
                // do nothing.
            }
        }

        return RepeatStatus.FINISHED;
    }
}
```

## 説明

項目番	説明
(1)	try-catchを実装する。
(2)	例外ハンドリングを実装する この例では、発生した例外のスタックトレースをログ出力している。
(3)	トランザクションをロールバックするため、例外を再度スローする。

### 6.2.2.2. ジョブ単位の例外ハンドリング

ジョブ単位に例外ハンドリング方法を説明する。

チャンクモデルとタスクレットモデルとで共通のハンドリング方法となる。

システム例外や致命的エラー等エラーはジョブ単位に [JobExecutionListener](#)インターフェースの実装を行う。

例外ハンドリング処理を集約して定義するために、ステップごとにハンドリング処理を定義はせずジョブ単位でハンドリングを行う。

ここでの例外ハンドリングは、ログ出力、およびExitCodeの設定を行い、トランザクション処理は実装しないこと。

#### トランザクション処理の禁止



[JobExecutionListener](#)で行われる処理は、業務トランザクション管理の範囲外となる。よってジョブ単位の例外ハンドリングでトランザクション処理を実施することは禁止する。

ここでは、ItemProcessorで例外が発生したときのハンドリング例を示す。リスナーの設定方法については、[リスナーの設定](#)を参照。

## ItemProcessorの実装例

```
@Component
public class AmountCheckProcessor implements
    ItemProcessor<SalesPerformanceDetail, SalesPerformanceDetail> {

    // omitted.

    private StepExecution stepExecution;

    // (1)
    @BeforeStep
    public void beforeStep(StepExecution stepExecution) {
        this.stepExecution = stepExecution;
    }

    @Override
    public SalesPerformanceDetail process(SalesPerformanceDetail item)
        throws Exception {
        // (2)
        try {
            checkAmount(item.getAmount(), amountLimit);
        } catch (ArithmetricException ae) {
            // (3)
            stepExecution.getExecutionContext().put("ERROR_ITEM", item);
            // (4)
            throw new IllegalStateException("check error at processor.", ae);
        }
        return item;
    }
}
```

## JobExecutionListenerでの例外ハンドリング実装

```
@Component
public class JobErrorLoggingListener implements JobExecutionListener {

    private static final Logger logger =
        LoggerFactory.getLogger(JobErrorLoggingListener.class);

    @Override
    public void beforeJob(JobExecution jobExecution) {
        // do nothing.
    }

    // (5)
    @Override
    public void afterJob(JobExecution jobExecution) {

        // whole job execution
        List<Throwable> exceptions = jobExecution.getAllFailureExceptions(); // (6)
        // (7)
        if (exceptions.isEmpty()) {
            return;
        }
        // (8)
        logger.info("This job has occurred some exceptions as follow. " +
            "[job-name:{}] [size:{}]",
            jobExecution.getJobInstance().getJobName(), exceptions.size());
        exceptions.forEach(th -> logger.error("exception has occurred in job.", th));

        // (9)
        jobExecution.getStepExecutions().forEach(stepExecution -> {
            Object errorItem = stepExecution.getExecutionContext()
                .get("ERROR_ITEM"); // (10)
            if (errorItem != null) {
                logger.error("detected error on this item processing. " +
                    "[step:{}] [item:{}]", stepExecution.getStepName(),
                    errorItem);
            }
        });
    }
}
```

### 説明

項目番号	説明
(1)	JobExecutionListenerでエラーデータを出力するため、ステップ実行前にStepExecutionインスタンスを取得する。
(2)	try-catchを実装する。

項目番	説明
(3)	例外ハンドリングを実装する この例では、 <code>StepExecution</code> インスタンスのコンテキストにエラーデータを <code>ERROR_ITEM</code> というキーで格納している。
(4)	<code>JobExecutionListener</code> で例外ハンドリングをするために、例外をスローする。
(5)	<code>afterJob</code> メソッドに例外ハンドリングを実装する。
(6)	引数の <code>jobExecution</code> からジョブ全体で発生したエラー情報を取得する。
(7)	エラー情報がない場合は、正常終了とする。
(8)	エラー情報がある場合は、例外ハンドリングを行う。 この例では、発生した例外をすべてスタックトレース付きのログ出力を行っている。
(9)	この例では、エラーデータがある場合はログ出力を行うようにしている。 ジョブで定義されたすべてのステップから <code>StepExecution</code> インスタンスを取得し、 <code>ERROR_ITEM</code> というキーでエラーデータが格納されているかチェックする。 格納されていた場合は、エラーデータとしてログ出力する。

*ExecutionContext*へ格納するオブジェクト



`ExecutionContext`へ格納するオブジェクトは、`java.io.Serializable`を実装したクラスでなければならない。これは、`ExecutionContext`が`JobRepository`へ格納されるためである。

### 6.2.2.3. 処理継続可否の決定

例外発生時にジョブの処理継続可否を決定する実装方法を説明する。

#### 処理継続可否方法一覧

- ・スキップ
- ・リトライ
- ・処理中断

#### 6.2.2.3.1. スキップ

エラーレコードをスキップして、処理を継続する方法を説明する。

#### チャンクモデル

チャンクモデルでは、各処理のコンポーネントで実装方法が異なる。



ここで説明する内容を適用する前に、必ず`<skippable-exception-classes>`を使わない理由についてを一読すること。

- ・`ItemReader`でのスキップ
- ・`ItemProcessor`でのスキップ
- ・`ItemWriter`でのスキップ

## ItemReaderでのスキップ

<batch:chunk>のskip-policy属性にスキップ方法を指定する。<batch:skippable-exception-classes>に、スキップ対象とするItemReaderで発生する例外クラスを指定する。  
skip-policy属性には、Spring Batchが提供している下記に示すいづれかのクラスを使用する。

### skip-policy一覧

クラス名	説明
AlwaysSkipItemSkipPolicy	常にスキップをする。
NeverSkipItemSkipPolicy	スキップをしない。
LimitCheckingItemSkipPolicy	指定したスキップ数の上限に達するまでスキップをする。 上限値に達した場合は、以下の例外が発生する。 <code>org.springframework.batch.core.step.skip.SkipLimitExceededException</code> skip-policyを省略した時にデフォルトで使われるスキップ方法である。
ExceptionClassifierSkipPolicy	例外ごとに適用するskip-policyを変えたい場合に利用する。

スキップの実装例を説明する。

FlatFileItemReaderでCSVファイルを読み込む際、不正なレコードが存在するケースを扱う。  
なお、この時以下の例外が発生する。

- `org.springframework.batch.item.ItemReaderException`(ベースとなる例外クラス)
  - `org.springframework.batch.item.file.FlatFileParseException`(発生する例外クラス)

skip-policy別に定義方法を示す。

## 前提とするItemReaderの定義

```
<bean id="detailCSVReader"
    class="org.springframework.batch.item.file.FlatFileItemReader" scope="step">
    <property name="resource" value="file:${jobParameters[inputFile]}"/>
    <property name="lineMapper">
        <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <property name="lineTokenizer">
                <bean
                    class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
                    p:names="branchId,year,month,customerId,amount"/>
            </property>
            <property name="fieldSetMapper">
                <bean
                    class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper"
                    p:targetType="org.terasoluna.batch.functionalttest.app.model.performance.SalesPerformanceDetail"/>
            </property>
        </bean>
    </property>
</bean>
```

## AlwaysSkipItemSkipPolicy

### AlwaysSkipItemSkipPolicyの指定例

```
<!-- (1) -->
<bean id="skipPolicy"
    class="org.springframework.batch.core.step.skip.AlwaysSkipItemSkipPolicy"/>

<batch:job id="jobSalesPerfAtSkipAllReadError" job-repository="jobRepository">
    <batch:step id="jobSalesPerfAtSkipAllReadError.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="detailCSVReader"
                processor="amountCheckProcessor"
                writer="detailWriter" commit-interval="10"
                skip-policy="skipPolicy" > <!-- (2) -->
            </batch:chunk>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

## 説明

項番	説明
(1)	AlwaysSkipItemSkipPolicyをBean定義する。
(2)	<batch:chunk>のskip-policy属性に(1)で定義したBeanを設定する。

## *NeverSkipItemSkipPolicy*

*NeverSkipItemSkipPolicy*の指定例

```
<!-- (1) -->
<bean id="skipPolicy"
      class="org.springframework.batch.core.step.skip.NeverSkipItemSkipPolicy"/>

<batch:job id="jobSalesPerfAtSkipNeverReadError" job-repository="jobRepository">
    <batch:step id="jobSalesPerfAtSkipNeverReadError.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="detailCSVReader"
                          processor="amountCheckProcessor"
                          writer="detailWriter" commit-interval="10"
                          skip-policy="skipPolicy"> <!-- (2) -->
                </batch:chunk>
            </batch:tasklet>
        </batch:step>
    </batch:job>
```

### 説明

項番	説明
(1)	<i>NeverSkipItemSkipPolicy</i> をBean定義する。
(2)	<batch:chunk>のskip-policy属性に(1)で定義したBeanを設定する。

## *LimitCheckingItemSkipPolicy*

## *LimitCheckingItemSkipPolicy*の指定例

```
(1)
<!--
<bean id="skipPolicy"
    class="org.springframework.batch.core.step.skip.LimitCheckingItemSkipPolicy"/>
-->

<batch:job id="jobSalesPerfAtValidSkipReadError" job-repository="jobRepository">
    <batch:step id="jobSalesPerfAtValidSkipReadError.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="detailCSVReader"
                processor="amountCheckProcessor"
                writer="detailWriter" commit-interval="10"
                skip-limit="2"> <!-- (2) -->
                <!-- (3) -->
                <batch:skippable-exception-classes>
                    <!-- (4) -->
                    <batch:include
                        class="org.springframework.batch.item.ItemReaderException"/>
                </batch:skippable-exception-classes>
            </batch:chunk>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

### 説明

項目番	説明
(1)	<i>LimitCheckingItemSkipPolicy</i> をBean定義する。 <i>skip-policy</i> 属性省略時のデフォルトであるため、定義しなくてもよい。
(2)	<batch:chunk>の <i>skip-limit</i> 属性にスキップ数の上限値を設定する。 <i>skip-policy</i> 属性はデフォルトを使用ため省略。
(3)	<batch:skippable-exception-classes>を定義し、タグ内に対象となる例外を設定する。
(4)	<i>ItemReaderException</i> をスキップ対象クラスとして設定を行う。

## *ExceptionClassifierSkipPolicy*

## ExceptionClassifierSkipPolicyの指定例

```
<!-- (1) -->
<bean id="skipPolicy"
      class="org.springframework.batch.core.step.skip.ExceptionClassifierSkipPolicy">
    <property name="policyMap">
      <map>
        <!-- (2) -->
        <entry key="org.springframework.batch.item.ItemReaderException"
               value-ref="alwaysSkip"/>
      </map>
    </property>
  </bean>
<!-- (3) -->
<bean id="alwaysSkip"
      class="org.springframework.batch.core.step.skip.AlwaysSkipItemSkipPolicy"/>

<batch:job id="jobSalesPerfAtValidNolimitSkipReadError"
            job-repository="jobRepository">
  <batch:step id="jobSalesPerfAtValidNolimitSkipReadError.step01">
    <batch:tasklet transaction-manager="jobTransactionManager">
      <!-- skip-limit value is dummy. -->
      <batch:chunk reader="detailCSVReader"
                    processor="amountCheckProcessor"
                    writer="detailWriter" commit-interval="10"
                    skip-policy="skipPolicy"> <!-- (4) -->
      </batch:chunk>
    </batch:tasklet>
  </batch:step>
</batch:job>
```

### 説明

項番	説明
(1)	ExceptionClassifierSkipPolicyをBean定義する。
(2)	policyMapプロパティに、キーを例外クラス、値をスキップ方法にしたマップを設定する。この例では、ItemReaderExceptionが発生したときに(3)で定義したスキップ方法になるように設定している。
(3)	例外別に実行したいスキップ方法を定義する。この例では、AlwaysSkipItemSkipPolicyを定義している。
(4)	<batch:chunk>のskip-policy属性に(1)で定義したBeanを設定する。

### ItemProcessorでのスキップ

ItemProcessor内でtry～catchをして、nullを返却する。

skip-policyによるスキップは、ItemProcessorで再処理が発生するため利用しない。詳細は、<skippable-exception-classes>を使わない理由についてを参照すること。

*ItemProcessor*における例外ハンドリングの制約



<skippable-exception-classes>を使わない理由についてにあるように、  
*ItemProcessor*では、<batch:skippable-exception-classes>を利用したスキップは禁  
止している。そのため、[コーディングポイント\(ItemProcessor編\)](#)で説明している  
「[ItemProcessListener](#)インターフェースを使用する方法」を応用したスキップはでき  
ない。

スキップの実装例を説明する。

[コーディングポイント\(ItemProcessor編\)](#)の *ItemProcessor*内でtry～catchする実装例を スキップに対応  
させる。

*ItemProcessor* 内でtry～catchする例

```
@Component
public class AmountCheckProcessor implements
    ItemProcessor<SalesPerformanceDetail, SalesPerformanceDetail> {

    // omitted.

    @Override
    public SalesPerformanceDetail process(SalesPerformanceDetail item) throws
Exception {
        // (1)
        try {
            checkAmount(item.getAmount(), amountLimit);
        } catch (ArithmaticException ae) {
            logger.warn("Exception occurred while processing. Skipped. [item:{}]",
                    item, ae); // (2)
            return null; // (3)
        }
        return item;
    }
}
```

説明

項番	説明
(1)	try～catchで実装する。
(2)	例外ハンドリングを実装する この例では、引数から取得した例外のスタックトレースをログ出力している。
(3)	nullを返却することでエラーデータをスキップする。

*ItemWriter*でのスキップ

*ItemWriter*においてスキップ処理は原則として行わない。

スキップが必要な場合でも、[skip-policy](#)によるスキップは、チャunkサイズが変動するので利用しな  
い。 詳細は、[<skippable-exception-classes>を使わない理由について](#)を参照すること。

## タスクレットモデル

ビジネスロジック内で例外をハンドリングし、独自にエラーレコードをスキップする処理を実装する。

タスクレットモデルにおける例外ハンドリングの [実装例](#)を スキップ対応させる。

## タスクレットモデルでの実装例

```
@Component
public class SalesPerformanceTasklet implements Tasklet {

    private static final Logger logger =
        LoggerFactory.getLogger(SalesPerformanceTasklet.class);

    // omitted

    @Override
    public RepeatStatus execute(StepContribution contribution,
                               ChunkContext chunkContext) throws Exception {

        // (1)
        try {
            reader.open(chunkContext.getStepContext().getStepExecution()
                .getExecutionContext());

            List<SalesPerformanceDetail> items = new ArrayList<>(10);
            SalesPerformanceDetail item = null;
            do {
                // Pseudo operation of ItemReader
                // omitted

                // Pseudo operation of ItemProcessor
                checkAmount(item.getAmount(), amountLimit);

                // Pseudo operation of ItemWriter
                // omitted

            } while (item != null);
        } catch (Exception e) {
            logger.warn("exception in tasklet. Skipped.", e); // (2)
            continue; // (3)
        } finally {
            try {
                reader.close();
            } catch (Exception e) {
                // do nothing.
            }
        }

        return RepeatStatus.FINISHED;
    }
}
```

## 説明

項目番	説明
(1)	try-catchを実装する。
(2)	例外ハンドリングを実装する この例では、発生した例外のスタックトレースをログ出力している。
(3)	continueにより、エラーデータの処理をスキップする。

#### 6.2.2.3.2. リトライ

例外を検知した場合に、規定回数に達するまで再処理する方法を説明する。

リトライには、状態管理の有無やリトライが発生するシチュエーションなどさまざまな要素を考慮する必要があり、確実な方法は存在しないうえに、むやみにリトライするとかえって状況を悪化させてしまう。

そのため、本ガイドラインでは、局所的なりトライを実現するorg.springframework.retry.support.RetryTemplateを利用する方法を説明する。



スキップと同様に<retryable-exception-classes>で対象となる例外クラスを指定する方法もある。しかし、<skippable-exception-classes>を使わない理由についてと同様に性能劣化を招く副作用があるため、TERASOLUNA Batch 5.xでは利用しない。

## RetryTemplate実装コード

```
public class RetryableAmountCheckProcessor implements
    ItemProcessor<SalesPerformanceDetail, SalesPerformanceDetail> {

    // omitted

    // (1)
    private RetryPolicy retryPolicy;

    @Override
    public SalesPerformanceDetail process(SalesPerformanceDetail item)
        throws Exception {

        // (2)
        RetryTemplate rt = new RetryTemplate();
        if (retryPolicy != null) {
            rt.setRetryPolicy(retryPolicy);
        }

        try {
            // (3)
            rt.execute(context -> {
                item.setAmount(item.getAmount().divide(new BigDecimal(10)));
                checkAmount(item.getAmount(), amountLimit);
                return null;
            });
        } catch (ArithmaticException ae) {
            // (4)
            throw new IllegalStateException("check error at processor.", ae);
        }
        return item;
    }

    public void setRetryPolicy(RetryPolicy retryPolicy) {
        this.retryPolicy = retryPolicy;
    }
}
```

## Bean定義

```
<!-- omitted -->

<bean id="amountCheckProcessor"
      class="org.terasoluna.batch.functionalttest.ch06.exceptionhandling.RetryableAmountCheck
      Processor"
      scope="step"
      p:retryPolicy-ref="retryPolicy"/> <!-- (5) -->

<!-- (6) (7) (8)-->
<bean id="retryPolicy" class="org.springframework.retry.policy.SimpleRetryPolicy"
      c:maxAttempts="3"
      c:retryableExceptions-ref="exceptionMap"/>

<!-- (9) -->
<util:map id="exceptionMap">
    <entry key="java.lang.ArithmetricException" value="true"/>
</util:map>

<batch:job id="jobSalesPerfWithRetryPolicy" job-repository="jobRepository">
    <batch:step id="jobSalesPerfWithRetryPolicy.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="detailCSVReader"
                         processor="amountCheckProcessor"
                         writer="detailWriter" commit-interval="10"/>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

## 説明

項番	説明
(1)	リトライ条件を格納する
(2)	RetryTemplateのインスタンスを作成する。 デフォルトは、リトライ回数=3、すべての例外がリトライ対象である。
(3)	RetryTemplate#executeメソッドで、リトライを行いたいビジネスロジックを実行するよう にする。 ビジネスロジック全体ではなく、リトライしたい部分のみをRetryTemplate#executeメソッド で実行するようにする。
(4)	リトライ回数が規定回数を超えた場合の例外ハンドリング。 ビジネスロジックで発生する例外がそのままスローされてくる。
(5)	(6)で定義するリトライ条件を指定する。
(6)	リトライ条件を、org.springframework.retry.RetryPolicyを実装したクラスで定義する。 この例では、Spring Batchから提供されているSimpleRetryPolicyを利用している。
(7)	コンストラクタ引数のmaxAttemptsにリトライ回数の指定をする。

項目番	説明
(8)	コンストラクタ引数の <code>retryableExceptions</code> に(9)で定義するリトライ対象の例外を定義したマップを指定する。
(9)	キーにリトライ対象の例外クラス、値に真偽値を設定したマップを定義する。 真偽値が <code>true</code> であれば、リトライ対象の例外となる。

#### 6.2.2.3.3. 処理中断

ステップ実行を打ち切りたい場合、スキップ・リトライ対象以外の`RuntimeException`もしくはそのサブクラスをスローする。

スキップの実装例を[LimitCheckingItemSkipPolicy](#)を元に示す。

*Bean*定義

```
<batch:job id="jobSalesPerfAtValidSkipReadError" job-repository="jobRepository">
    <batch:step id="jobSalesPerfAtValidSkipReadError.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="detailCSVReader"
                processor="amountCheckProcessor"
                writer="detailWriter" commit-interval="10"
                skip-limit="2">
                <batch:skippable-exception-classes>
                    <!-- (1) -->
                    <batch:include
                        class="org.springframework.batch.item.validator.ValidationException"/>
                </batch:skippable-exception-classes>
            </batch:chunk>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

説明

項目番	説明
(1)	<code>ValidationException</code> 以外の例外が発生すれば処理が中断される。

リトライの実装例を[リトライ](#)を元に示す。

## Bean定義

```
<!-- omitted -->

<bean id="retryPolicy" class="org.springframework.retry.policy.SimpleRetryPolicy"
    c:maxAttempts="3"
    c:retryableExceptions-ref="exceptionMap"/>

<util:map id="exceptionMap">
    <!-- (1) -->
    <entry key="java.lang.UnsupportedOperationException" value="true"/>
</util:map>

<batch:job id="jobSalesPerfWithRetryPolicy" job-repository="jobRepository">
    <batch:step id="jobSalesPerfWithRetryPolicy.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="detailCSVReader"
                processor="amountCheckProcessor"
                writer="detailWriter" commit-interval="10"/>
        </batch:tasklet>
    </batch:step>
</batch:job>
```

## 説明

項目番号	説明
(1)	UnsupportedOperationException以外の例外が発生すれば処理が中断される。

## 6.2.3. Appendix

### 6.2.3.1. <skippable-exception-classes>を使わない理由について

Spring Batchでは、ジョブ全体を対象としてスキップする例外を指定し、例外が発生したアイテムへの処理をスキップして処理を継続させる機能を提供している。

その機能は、以下のように<chunk>タグ配下に<skippable-exception-classes>タグを設定し、スキップ対象の例外を指定する形で実装する。

## <skippable-exception-classes>の使用例

```
<job id="flowJob">
  <step id="retryStep">
    <tasklet>
      <chunk reader="itemReader" writer="itemWriter"
        processor="itemProcessor" commit-interval="20"
        skip-limit="10">
        <skippable-exception-classes>
          <!-- specify exceptions to the skipped -->
          <include class="java.lang.Exception"/>
          <exclude class="java.lang.NullPointerException"/>
        </skippable-exception-classes>
      </chunk>
    </tasklet>
  </step>
</job>
```

この機能を利用することによって、入力チェックエラーが発生したレコードをスキップして後続データの処理を継続することは可能だが、TERASOLUNA Batch 5.xでは以下の理由により使用しない。

- <skippable-exception-classes>タグを利用して例外をスキップした場合、1つのチャンクに含まれるデータ件数が変動するため、性能劣化を引き起こす可能性がある。
- これは、例外の発生箇所(ItemReader/ItemProcessor/ItemWriter)によって変わる。詳細は後述する。



<skippable-exception-classes>を定義せずにSkipPolicyを利用することは必ず避ける  
暗黙的にすべての例外が登録された状況になり、性能劣化の可能性が飛躍的に高まる。

例外発生箇所(ItemReader/ItemProcessor/ItemWriter)ごとの挙動についてそれぞれ説明する。

なお、トランザクションの動作は例外の発生箇所によらず、例外が発生した場合は必ずロールバックした後、再度処理される。

### ItemReaderで例外が発生した場合

- ItemReaderの処理内で例外が発生した場合は、次のitemへ処理対象が移る。
- これによる副作用はない

### ItemProcessorで例外が発生した場合

- ItemProcessorの処理内で例外が発生した場合は、チャンクの最初に戻り1件目から再処理する。
- 再処理の対象にスキップされるitemは含まれない。
- 1度目の処理と再処理時のチャンクサイズは変わらない。

### ItemWriterで例外が発生した場合

- ItemWriterの処理内で例外が発生した場合は、チャンクの最初に戻り1件目から再処理する。
- 再処理はChunkSize=1に固定し、1件ずつ実行される。

- 再処理対象にスキップされるitemも含まれる。

`ItemProcessor`にて例外が発生した場合、`ChunkSize=1000`の場合を例に考えると、1000件目で例外が発生すると1件目から再処理が行われ、合計で1999件分の処理が実行されてしまう。

`ItemWriter`にて例外が発生した場合、`ChunkSize=1`に固定し再処理される。仮に`ChunkSize=1000`の場合を例に考えると、本来1回のトランザクションにも関わらず1000回のトランザクションに分割し処理されてしまう。

これらはジョブ全体の処理時間が長期化することを意味し、異常時に状況を悪化させる可能性が高い。また、二重処理すること自体が問題化する可能性を秘めており、設計製造に追加の考慮事項を産む。

よって、`<skippable-exception-classes>`を使用することは推奨しない。`ItemReader`でエラーになったデータをスキップすることはこれらの問題を引き起こさないが、事故を未然に防ぐためには基本的に避けるようにし、どうしても必要な場合に限定的に適用すること。

## 6.3. 処理の再実行

### 6.3.1. Overview

障害発生などに起因してジョブが異常終了した後に、ジョブを再実行することで回復する手段について説明する。

本機能は、チャンクモデルとタスクレットモデルとで使い方が異なるため、それぞれについて説明する。

ジョブの再実行には、以下の方法がある。

1. ジョブのリラン
2. ジョブのリストート
  - ステートレスリストート
    - 件数ベースリストート
    - ステートフルリストート
      - 処理状態を判断し、未処理のデータを抽出して処理するリストート
      - 処理状態を識別するための処理を別途実装する必要がある

以下に用語を定義する。

リラン

ジョブを最初からやり直すこと。

事前作業として、データ初期化など障害発生前のジョブ開始時点に状態を回復する必要がある。

リストート

ジョブが中断した箇所から処理を再開すること。

処理再開位置の保持・取得方法、再開位置までのデータスキップ方法などをあらかじめ設計/実装する必要がある。

リストートには、ステートレスとステートフルの2種類がある。

ステートレスリストート

個々の入力データに対する状態(未処理/処理済)を考慮しないリストート方法。

件数ベースリストート

ステートレスリストートの1つ。

処理した入力データ件数を保持し、リストート時にその件数分入力データをスキップする方法。

出力が非トランザクショナルなリソースの場合は、出力位置を保持し、リストート時にその位置まで書き込み位置を移動することも必要になる。

ステートフルリストート

個々の入力データに対する状態(未処理/処理済)を判断し、未処理のデータのみを取得条件とするリストート方法。

出力が非トランザクショナルなリソースの場合は、リソースを追記可能にして、リストート時には前回の結果へ追記していくようにする。

一般的に、再実行の方法はリランがもっとも簡単である。リラン < ステートレスリストア < ステートフルリストアの順に、設計や実装が難しくなる。無論、可能であれば常にリランとすることが好ましいが、ユーザが実装するジョブ1つ1つに対して、許容するバッチウィンドウや処理特性に応じてどの方法を適用するか検討してほしい。

### 6.3.2. How to use

リランとリストアの実現方法について説明する。

#### 6.3.2.1. ジョブのリラン

ジョブのリランを実現方法する説明する。

1. リラン前にデータの初期化などデータ回復の事前作業を実施する。
2. 失敗したジョブを同じ条件(同じパラメータ)で再度実行する。
  - Spring Batchでは同じパラメータでジョブを実行すると二重実行と扱われるが、TERASOLUNA Batch 5.xでは別ジョブとして扱う。  
詳細は、[パラメータ変換クラスについて](#)を参照のこと。

#### 6.3.2.2. ジョブのリストア

ジョブのリストア方法を説明する。

ジョブのリストアを行う場合は、同期実行したジョブに対して行うことを基本とする。

非同期実行したジョブは、リストアではなくリランで対応するジョブ設計にすることを推奨する。これは、「意図したリストア実行」なのか「意図しない重複実行」であるかの判断が難しく、運用で混乱をきたす可能性があるからである。

非同期実行ジョブでリストア要件がどうしても外せない場合は、「意図したリストア実行」を明確にするために、以下の方法を利用できる。

- [CommandLineJobRunner](#)の[-restart](#)によるリストア
  - 非同期実行したジョブを別途同期実行によりリストアする。逐次で回復処理を進めていく際に有効となる。
- [JobOperator#restart\(JobExecutionId\)](#)によるリストア
  - 非同期実行したジョブを、再度非同期実行の仕組み上でリストアする。一括で回復処理を進めていく際に有効となる。
    - [非同期実行\(DBポーリング\)](#)はリストアをサポートしていない。そのため、別途ユーザにて実装する必要がある。
    - [非同期実行\(Webコンテナ\)](#)はリストアの実現方法をガイドしている。この記述にしたがって、ユーザにて実装すること。

入力チェックがある場合のリスタートについて

入力チェックエラーは、チェックエラーの原因となる入力リソースを修正しない限り回復不可能である。参考までに、入力チェックエラーが発生した際の入力リソース修正例を以下に示す。

1. 入力チェックエラーが発生した場合は、対象データが特定できるようにログ出力を行う。
2. 出力されたログ情報にもとづいて、入力データの修正を行う。
  - 入力データの順番が変わらないようにする。
  - 修正方法は入力リソースの生成方法により対応が異なる。
    - 手動で修正
    - ジョブなどで再作成
    - 連携元からの再送
3. 修正した入力データを配備して、リスタートを実行する。

多重処理(*Partition Step*)の場合について

**i** *多重処理(Partition Step)*でリスタートする場合、再び分割処理から処理が実施される。データを分割した結果、すべて処理済みであった場合、無駄な分割処理が行われ*JobRepository*上には記録されるが、これによるデータ不整合などの問題は発生しない。

### 6.3.2.3. ステートレスリスタート

ステートレスリスタートを実現する方法を説明する。

TERASOLUNA Batch 5.xでのステートレスリスタートは、件数ベースのリスタートを指す。これは、Spring Batchの仕組みをそのまま利用することで実現する。

件数ベースのリスタートは、チャンクモデルのジョブ実行で使用できる。また、件数ベースのリスタートは、*JobRepository*に登録される入出力に関するコンテキスト情報を利用する。よって、件数ベースのリスタートでは、*JobRepository*はインメモリデータベースではなく、永続性が担保されているデータベースを使用することを前提とする。

*JobRepository*の障害発生時について

**!** *JobRepository*への更新は、ユーザが使用するデータベースのトランザクションとは独立したトランザクションで行われる。つまり、業務処理に対する障害のみがリカバリ対象となる。これは、*JobRepository*に障害が発生した場合は実際の処理件数とずれる可能性があり、リスタート時に二重処理の危険性があることを意味する。よって、*JobRepository*の可用性を検討したり、次点の方法としてリランの方法を検討しておいたりといった、障害時の対処方法を検討する必要がある。

### リスタート時の入力

Spring Batchが提供しているItemReaderのほとんどが件数ベースのリスタートに対応しているため、特別な対応は不要である。

件数ベースのリスタート可能なItemReaderを自作する場合は、リスタート処理が実装されている以下の抽象クラスを拡張すればよい。

- `org.springframework.batch.item.support.AbstractItemCountingItemStreamItemReader`

件数ベースリスタートは、あくまで件数のみを基準としてリスタート開始点を決定するため、入力データの変更/追加/削除を検出することができない。ジョブが異常終了した後、回復するために入力データを補正することはしばしばあるが、以下のようなデータの変更を行った場合は、ジョブが正常終了した結果と、ジョブが異常終了した後リスタートして回復できた結果、の間で出力に差が出るため注意すること。

- データの取得順を変更する
  - リスタート時に、重複処理や未処理となるデータが発生してしまい、リランした結果と異なる回復結果になるため、決して行ってはいけない。
- 処理済みデータを更新する
  - リスタート時に更新したデータは読み飛ばされるので、リランした結果とリスタートした結果で回復結果が変わるために好ましくない場合がある。
- 未処理データを更新または追加する
  - リランした結果と同じ回復結果になるため許容する。ただし、初回実行で正常終了した結果とは異なる。これは異常なデータを緊急対処的にパッチする場合や、実行時点で受領したデータを可能な限り多く処理する際に限定して使うとよい。

## リスタート時の出力

非トランザクショナルなリソースへの出力には注意が必要である。たとえば、ファイルではどの位置まで出力していたかを把握し、その位置から出力を行わなければいけない。

Spring Batchが提供している`FlatFileItemWriter`は、コンテキストから前回の出力位置を取得して、リスタート時にはその位置から出力を行ため、特別な対応は不要である。

トランザクショナルなリソースについては、失敗時にロールバックが行われているため、リスタート時には特に対処することなく処理を行うことができる。

上記の条件を満たしていれば、失敗したジョブに`-restart`のオプションを付加して再度実行すればよい。以下にジョブのリスタート例を示す。

## 同期実行したジョブのリスタート例

```
# (1)
java -cp dependency/*
org.springframework.batch.core.launch.support.CommandLineJobRunner <jobPath> <jobName>
-restart
```

## 説明

項目番号	説明
------	----

(1)	<code>CommandLineJobRunner</code> へ失敗したジョブと同じジョブBeanのパスとジョブ名を指定し、 <code>-restart</code> を付加して実行する。 ジョブパラメータは、 <code>JobRepository</code> から復元されるため指定は不要。
-----	---

非同期実行(DBポーリング)で実行したジョブのリスタート例を以下に示す。

非同期実行(DBポーリング)で実行したジョブのリスタート例

```
# (1)
java -cp dependency/*
org.springframework.batch.core.launch.support.CommandLineJobRunner <JobExecutionId>
-restart
```

#### 説明

項番	説明
(1)	<code>CommandLineJobRunner</code> へ失敗したジョブと同じジョブ実行ID(JobExecutionId)を指定し、 <code>-restart</code> を付加して実行する。 ジョブパラメータは、 <code>JobRepository</code> から復元されるため指定は不要。 ジョブ実行IDは、ジョブ要求テーブルから取得することができる。 ジョブ要求テーブルについては、 <a href="#">ポーリングするテーブルについて</a> を参照。

ジョブ実行IDのログ出力



異常終了したジョブのジョブ実行IDを迅速に特定するため、ジョブ終了時や例外発生時にジョブ実行IDをログ出力するリスナーや例外ハンドリングクラスを実装することを推奨する。

非同期実行(Webコンテナ)でのリスタート例を以下に示す。

非同期実行(Webコンテナ)で実行したジョブのリスタート例

```
public long restart(long JobExecutionId) throws Exception {
    return jobOperator.restart(JobExecutionId); // (1)
}
```

#### 説明

項番	説明
(1)	<code>JobOperator</code> へ失敗したジョブと同じジョブ実行ID(JobExecutionId)を指定し、 <code>restart</code> メソッドで実行する。 ジョブパラメータは、 <code>JobRepository</code> から復元される。 ジョブ実行IDは、WebAPでジョブ実行した際に取得したIDを利用するか、 <code>JobRepository</code> から取得することができる。 取得方法は、 <a href="#">ジョブの状態管理</a> を参照。

#### 6.3.2.4. ステートフルリスタート

ステートフルリスタートを実現する方法を説明する。

ステートフルリスタートとは、実行時に入出力結果を付きあわせて未処理データだけ取得することで再処

理する方法である。この方法は、状態保持・未処理判定など設計が難しいが、データの変更に強い特徴があるため、時々用いられることがある。

ステートフルリストアでは、リストア条件を入出力リソースから判定するため、[JobRepository](#)の永続化は不要となる。

#### リストア時の入力

入出力結果を付きあわせて未処理データだけ取得するロジックを実装したItemReaderを用意する。

#### リストア時の出力

[ステートレスリストア](#)と同様に非トランザクショナルなリソースへ出力には注意が必要になる。

ファイルの場合、コンテキストを使用しないことを前提にすると、ファイルの追記を許可するような設計が必要になる。

ステートフルリストアは、[ジョブのリラン](#)と同様に失敗時のジョブと同じ条件でジョブを再実行する。ステートレスリストアとは異なり、[-restart](#)のオプションは使用しない。

簡単ステートフルなリストアの実現例を下記に示す。

#### 処理仕様

1. 入力対象のテーブルに処理済カラムを定義し、処理が成功したらNULL以外の値で更新する。
  - 未処理データの抽出条件は、処理済カラムの値がNULLとなる。
2. 処理結果をファイルに出力する。

### *RestartOnConditionRepository.xml*

```
<!-- (1) -->
<select id="findByProcessedIsNull"

resultType="org.terasoluna.batch.functionaltest.app.model.plan.SalesPlanDetail">
<![CDATA[
SELECT
    branch_id AS branchId, year, month, customer_id AS customerId, amount
FROM
    sales_plan_detail
WHERE
    processed IS NULL
ORDER BY
    branch_id ASC, year ASC, month ASC, customer_id ASC
]]>
</select>

<!-- (2) -->
<update id="update"
parameterType="org.terasoluna.batch.functionaltest.app.model.plan.SalesPlanDetail">
<![CDATA[
UPDATE
    sales_plan_detail
SET
    processed = '1'
WHERE
    branch_id = #{branchId}
AND
    year = #{year}
AND
    month = #{month}
AND
    customer_id = #{customerId}
]]>
</update>
```

### *restartOnConditionBasisJob.xml*

```
<!-- (3) -->
<bean id="reader" class="org.mybatis.spring.batch.MyBatisCursorItemReader"

p:queryId="org.terasoluna.batch.functionaltest.ch06.reprocessing.repository.RestartOnConditionRepository.findByZeroOrLessAmount"
    p:sqlSessionFactory-ref="jobSqlSessionFactory"/>

<!-- (4) -->
<bean id="dbWriter" class="org.mybatis.spring.batch.MyBatisBatchItemWriter"

p:statementId="org.terasoluna.batch.functionaltest.ch06.reprocessing.repository.Restar
```

```

tOnConditionRepository.update"
    p:sqlSessionTemplate-ref="batchModeSqlSessionTemplate"/>

<bean id="fileWriter"
    class="org.springframework.batch.item.file.FlatFileItemWriter" scope="step"
    p:resource="file:#{jobParameters[outputFile]}"
    p:appendAllowed="true"> <!-- (5) -->
<property name="lineAggregator">
    <bean
class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
        <property name="fieldExtractor">
            <bean
class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor"
                p:names="branchId,year,month,customerId,amount"/>
        </property>
    </bean>
</property>
</bean>
<!-- (6) -->
<bean id="compositeWriter"
class="org.springframework.batch.item.support.CompositeItemWriter">
    <property name="delegates">
        <list>
            <ref bean="fileWriter"/>
            <ref bean="dbWriter"/>
        </list>
    </property>
</bean>

<batch:job id="restartOnConditionBasisJob"
    job-repository="jobRepository" restartable="false"> <!-- (7) -->

    <batch:step id="restartOnConditionBasisJob.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <batch:chunk reader="reader" processor="amountUpdateItemProcessor"
                writer="compositeWriter" commit-interval="10" />
        </batch:tasklet>
    </batch:step>

</batch:job>

```

#### リスタートのコマンド実行例

```

# (8)
java -cp dependency/*
org.springframework.batch.core.launch.support.CommandLineJobRunner <jobPath> <jobName>
<jobParameters> ...

```

#### 説明

項目番	説明
(1)	処理済カラムがNULLのデータのみ抽出するようにSQLを定義する。
(2)	処理済カラムをNULL以外で更新するSQLを定義する。
(3)	ItemReaderには、(1)で定義したSQLIDを設定する。
(4)	データベースへ更新は、(2)で定義したSQLIDを設定する。
(5)	リスタート時に前回中断箇所から書き込み可能にするため、ファイルの追記を許可する。
(6)	ファイル出力 → データベース更新の順序で処理されるようにCompositeItemWriterを定し、chunkのwriterに設定する。
(7)	必須ではないが、誤って-restartオプションをつけて起動された場合にエラーになるようにrestartable属性をfalseに設定しておく。
(8)	失敗したジョブの実行条件で再度実行を行う。

ジョブのrestartable属性について



restartableがtrueの場合、ステートレスリスタートで説明したとおり、コンテキスト情報を使い入出力データの読み飛ばしを行う。ステートフルリスタートでSpring Batch提供のItemReaderやItemWriterを使用している場合、この動作により期待した処理が行われなくなる可能性がある。そのため、restartableをfalseにすることで、-restartオプションによる起動はエラーとなり、誤動作を防止することができる。

# Chapter 7. ジョブの管理

## 7.1. Overview

ジョブの実行を管理する方法について説明する。

本機能は、チャンクモデルとタスクレットモデルとで同じ使い方になる。

### 7.1.1. ジョブの実行管理とは

ジョブの起動状態や実行結果を記録しバッチシステムを維持することを指す。特に、異常発生時の検知や次に行うべき行動(異常終了後のリラン・リストート等)を判断するために、必要な情報を確保することが重要である。

バッチアプリケーションの特性上、起動直後にその結果をユーザインターフェースで確認できることは稀である。よって、ジョブスケジューラ/RDBMS/アプリケーションログといった、ジョブの実行とは別に実行状態・結果の記録を行う仕組みが必要となる。

#### 7.1.1.1. Spring Batch が提供する機能

Spring Batchは、ジョブの実行管理向けに以下のインターフェースを提供している。

ジョブの管理機能一覧

機能	対応するインターフェース
ジョブの実行状態・結果の記録	<a href="#">org.springframework.batch.core.repository.JobRepository</a>
ジョブの終了コードとプロセス終了コードの変換	<a href="#">org.springframework.batch.core.launch.support.ExitCodeMapper</a>

Spring Batch はジョブの起動状態・実行結果の記録に[JobRepository](#)を使用する。TERASOLUNA Batch 5.xでは、以下のすべてに該当する場合は永続化は任意としてよい。

- 同期型ジョブ実行のみでTERASOLUNA Batch 5.xを使用する。
- ジョブの停止・リストートを含め、ジョブの実行管理はすべてジョブスケジューラに委ねる。
  - Spring Batchがもつ[JobRepository](#)を前提としたリストートを利用しない。

これらに該当する場合は[JobRepository](#)が使用するRDBMSの選択肢として、インメモリ・組み込み型データベースである[H2](#)を利用する。

一方で非同期実行を利用する場合や、Spring Batchの停止・リストートを活用する場合は、ジョブの実行状態・結果を永続化可能なRDBMSが必要となる。

デフォルトのトランザクション分離レベル

Spring Batchが提供するxsdでは、[JobRepository](#)のトランザクション分離レベルは[SERIALIZABLE](#)をデフォルト値としている。しかし、この場合、同期/非同期にかかわらず複数のジョブを同時に実行した際に[JobRepository](#)の更新で例外が発生してしまう。そのため、TERASOLUNA Batch 5.xでは、あらかじめ[JobRepository](#)のトランザクション分離レベルを[READ\\_COMMITTED](#)に設定している。





インメモリJobRepositoryの選択肢

Spring Batch にはインメモリでジョブの実行管理を行う

`org.springframework.batch.core.repository.support.MapJobRepositoryFactoryBean`

が用意されているが、本ガイドラインでは使用しない。

このクラスのJavadocにあるとおり、`This repository is only really intended for use in testing and rapid prototyping.` とテスト用であることが示されており、ま

た、`Not suited for use in multi-threaded jobs with splits` と並列処理には不適

切であると示されているためである。

ジョブスケジューラを使用したジョブの実行管理については各製品のマニュアルを参照のこと。

本ガイドラインではTERASOLUNA Batch 5.x内部でJobRepositoryを用いたジョブの状態を管理するうえで関連する、以下の項目について説明する。

TERASOLUNA Batch内部での状態管理に関する項目

- [ジョブの状態管理](#)

- 状態を永続化する方法
- 状態を確認する方法
- ジョブを手動停止する方法

- 終了コードのカスタマイズ

- 二重起動防止

- ロギング

- メッセージ管理

## 7.2. How to use

JobRepositoryはSpring Batchにより自動的にRDBMSの新規登録・更新が行われる。

ジョブの状態・実行結果の確認を行う場合は、意図しない変更処理がバッチアプリケーションの内外から行われることのないよう、以下のいずれかの方法を選択する。

- [ジョブの状態管理](#)に関するテーブルに対しクエリを発行する
- `org.springframework.batch.core.explore.JobExplorer`を使用する

### 7.2.1. ジョブの状態管理

JobRepositoryを用いたジョブの状態管理方法を説明する。

Spring Batchにより、以下のEntityがRDBMSのテーブルに登録される。

JobRepositoryで管理されるEntityクラスとテーブル名

項目番号	Entityクラス	テーブル名	生成単位	説明
(1)	<code>JobExecution</code>	<code>BATCH_JOB_EXECUTION</code>	1回のジョブ実行	ジョブの状態・実行結果を保持する。
(2)	<code>JobExecutionContext</code>	<code>BATCH_JOB_EXECUTION_CONTEXT</code>	1回のジョブ実行	ジョブ内部のコンテキストを保持する。

項目番	Entityクラス	テーブル名	生成単位	説明
(3)	JobExecutionParams	BATCH_JOB_EXECUTION_PARAMS	1回のジョブ実行	起動時に与えられたジョブパラメータを保持する。
(4)	StepExecution	BATCH_STEP_EXECUTION	1回のステップ実行	ステップの状態・実行結果、コミット・ロールバック件数を保持する。
(5)	StepExecutionContext	BATCH_STEP_EXECUTION_CONTEXT	1回のステップ実行	ステップ内部のコンテキストを保持する。
(6)	JobInstance	BATCH_JOB_INSTANCE	ジョブ名とジョブパラメータの組み合わせ	ジョブ名、およびジョブパラメータをシリアル化した文字列を保持する。

たとえば、1回のジョブ起動で3つのステップを実行した場合、以下の差が生じる

- JobExecution、JobExecutionContext、JobExecutionParamsは1レコード登録される
- StepExecution、StepExecutionContextは3レコード登録される

また、JobInstanceは過去に起動した同名ジョブ・同一パラメータによる二重実行を抑止するために使用されるが、TERASOLUNA Batch 5.xではこのチェックを行わない。詳細は[二重起動防止](#)を参照。



JobRepositoryによる各テーブルの構成は、Spring Batchのアーキテクチャにて説明している。

チャンク方式における*StepExecution*の件数項目について  
以下のように、不整合が発生しているように見えるが、仕様上妥当なケースがある。

- *StepExecution(BATCH\_STEP\_EXECUTIONテーブル)*のトランザクション発行回数が入力データ件数と一致しない場合がある。
  - トランザクション発行回数は*BATCH\_STEP\_EXECUTION*の*COMMIT\_COUNT*と*ROLLBACK\_COUNT*の総和を指す。  
ただし、入力データ件数がチャンクサイズで割り切れる場合*COMMIT\_COUNT*が+1となる。  
これは入力データ件数分を読み込んだ後、終端を表すnullも入力データとカウントされて空処理されるためである。
- *BATCH\_STEP\_EXECUTION*と*BATCH\_STEP\_EXECUTION\_CONTEXT*の処理件数が異なることがある。
  - *BATCH\_STEP\_EXECUTION*テーブルの*READ\_COUNT*、*WRITE\_COUNT*はそれぞれ*ItemReader*、*ItemWriter*による読み込み・書き込みを行った件数が記録される。
  - *BATCH\_STEP\_EXECUTION\_CONTEXT*テーブルの*SHORT\_CONTEXT*カラムはJSON形式で*ItemReader*による読み込み処理件数が記録される。しかし、必ずしも*BATCH\_STEP\_EXECUTION*による処理件数と一致しない。
  - これはチャンク方式による*BATCH\_STEP\_EXECUTION*テーブルが成功・失敗を問わず読み込み・書き込み件数を記録するのに対し、*BATCH\_STEP\_EXECUTION\_CONTEXT*テーブルは処理途中で失敗した場合のリストアで再開される位置として記録するためである。

#### 7.2.1.1. 状態の永続化

外部RDBMSを使用することで*JobRepository*によるジョブの実行管理情報を永続化させることができる。*batch-application.properties*の以下項目を外部RDBMS向けのデータソース、スキーマ設定となるよう修正する。

##### *batch-application.properties*

```
# (1)
# Admin DataSource settings.
admin.jdbc.driver=org.postgresql.Driver
admin.jdbc.url=jdbc:postgresql://serverhost:5432/admin
admin.jdbc.username=postgres
admin.jdbc.password=postgres

# (2)
spring-batch.schema.script=classpath:org/springframework/batch/core/schema-
postgresql.sql
```

##### 設定内容の項目一覧(PostgreSQL)

項目番	説明
(1)	接頭辞adminが付与されているプロパティの値として、接続する外部RDBMSの設定をそれぞれ記述する。
(2)	アプリケーション起動時にJobRepositoryとしてスキーマの自動生成を行うスクリプトファイルを指定する。

#### 管理用/業務用データソースの補足



- DBへの接続設定は、管理用と業務用データソースとして別々に定義する。TERASOLUNA Batch 5.xでは別々に定義した上で、JobRepositoryは、プロパティ接頭辞にadminが付与された管理用データソースを使用するよう設定済みである。
- 非同期実行(DBポーリング)を使用する場合は、ジョブ要求テーブルも同じ管理用データソース、スキーマ生成スクリプトを指定すること。  
詳細は[非同期実行\(DBポーリング\)](#)を参照。

#### 7.2.1.2. ジョブの状態・実行結果の確認

JobRepositoryからジョブの実行状態を確認する方法について説明する。  
いずれの方法も、あらかじめ確認対象のジョブ実行IDが既知であること。

##### 7.2.1.2.1. クエリを直接発行する

RDBMSコンソールを用い、JobRepositoryが永続化されたテーブルに対して直接クエリを発行する。

SQLサンプル

```
admin=# select JOB_EXECUTION_ID, START_TIME, END_TIME, STATUS, EXIT_CODE from
BATCH_JOB_EXECUTION where JOB_EXECUTION_ID = 1;
+-----+-----+-----+-----+
| job_execution_id | start_time | end_time | status |
| exit_code        |
+-----+-----+-----+-----+
1 | 2017-02-14 17:57:38.486 | 2017-02-14 18:19:45.421 | COMPLETED |
COMPLETED
(1 row)
admin=# select JOB_EXECUTION_ID, STEP_EXECUTION_ID, START_TIME, END_TIME, STATUS,
EXIT_CODE from BATCH_STEP_EXECUTION where JOB_EXECUTION_ID = 1;
+-----+-----+-----+-----+
| job_execution_id | step_execution_id | start_time | end_time |
| status | exit_code |
+-----+-----+-----+-----+
1 | 1 | 2017-02-14 17:57:38.524 | 2017-02-14 18:19:45
.41 | COMPLETED | COMPLETED
(1 row)
```

##### 7.2.1.2.2. JobExplorerを利用する

バッチアプリケーションと同じアプリケーションコンテキストを共有可能な環境下で、JobExplorerをインジェクションすることでジョブの実行状態を確認する。

## APIコードサンプル

```
// omitted.

@Inject
private JobExplorer jobExplorer;

private void monitor(long jobExecutionId) {

    // (1)
    JobExecution jobExecution = jobExplorer.getJobExecution(jobExecutionId);

    // (2)
    String jobName = jobExecution.getJobInstance().getJobName();
    Date jobStartTime = jobExecution.getStartTime();
    Date jobEndTime = jobExecution.getEndTime();
    BatchStatus jobBatchStatus = jobExecution.getStatus();
    String jobExitCode = jobExecution.getExitStatus().getExitCode();

    // omitted.

    // (3)
    jobExecution.getStepExecutions().forEach( s -> {
        String stepName = s.getStepName();
        Date stepStartTime = s.getStartTime();
        Date stepEndTime = s.getEndTime();
        BatchStatus stepStatus = s.getStatus();
        String stepExitCode = s.getExitStatus().getExitCode();

        // omitted.
    });
}
}
```

## 設定内容の項目一覧(PostgreSQL)

項目番	説明
(1)	インジェクションされたJobExplorerからジョブ実行IDを指定しJobExecutionを取得する。
(2)	JobExecutionによるジョブの実行結果を取得する。
(3)	JobExecutionから、ジョブ内で実行されたステップのコレクションを取得し、個々の実行結果を取得する。

### 7.2.1.3. ジョブの停止

ジョブの停止とはJobRepositoryの実行中ステータスを停止中ステータスに更新し、ステップの境界や チャンク方式によるチャンクコミット時にジョブを停止させる機能である。

リストアと組み合わせることで、停止された位置からの処理を再開させることができる。



リストアの詳細はジョブのリストアを参照。

「ジョブの停止」は仕掛けり中のジョブを直ちに中止する機能ではなく、[JobRepository](#)の実行中ステータスを停止中に更新する機能である。ジョブに対して即座に仕掛けり中スレッドに対して割り込みするといったような、何らかの停止処理を行うわけではない。



このため、ジョブの停止は「チャンクの切れ目など、節目となる処理が完了した際に停止するよう予約する」ことともいえる。たとえば以下の状況下でジョブ停止を行っても、期待する動作とはならない。

- 単一ステップで[Tasklet](#)により構成されたジョブ実行。
- チャンク方式で、データ入力件数 < [commit-interval](#) のとき。
- 処理内で無限ループが発生している場合。

以下、ジョブの停止方法を説明する。

- コマンドラインからの停止
  - 同期型ジョブ・非同期型ジョブのどちらでも利用できる
  - [CommandLineJobRunner](#)の[-stop](#)を利用する

起動時のジョブ名を指定する方法

```
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
  classpath:/META-INF/jobs/job01/job01.xml job01 -stop
```

- ジョブ名指定によるジョブ停止は同名のジョブが並列で起動することが少ない同期バッチ実行時に適している。

ジョブ実行ID(*jobExecutionId*)を指定する方法

```
$ java org.springframework.batch.core.launch.support.CommandLineJobRunner \
  classpath:/META-INF/jobs/job01/job01.xml 3 -stop
```

- ジョブ実行ID指定によるジョブ停止は同名のジョブが並列で起動することの多い非同期バッチ実行時に適している。
  - JobExecutionIdの確認方法は[ジョブの状態・実行結果の確認](#)を参照のこと。
  - ジョブ実行IDを元にジョブ停止を行う場合は[JobOperation#stop\(\)](#)を利用してもよい。[JobOperation#stop\(\)](#)を用いたジョブの停止は[非同期実行ジョブの停止とリストア](#)を参照のこと。



## 7.2.2. 終了コードのカスタマイズ

同期実行によりジョブが終了した際、javaプロセスの終了コードをジョブの終了状態に応じてカスタマイズできる。終了コードのカスタマイズには以下2つの作業が必要となる。

- ジョブの終了状態を表す、ジョブの終了コードを変更する。
- ジョブ・ステップの終了コード(文字列)と、プロセスの終了コード(数値)をマッピングする。

以下順に説明する。

### 7.2.2.1. ジョブの終了コードを変更する。

文字列として返却されるジョブの終了コードを変更できる。

- ステップ終了時に任意の終了状態を返却するように[afterStep](#)メソッドを実装する。
  - [StepExecutionListener](#)を実装する。

#### StepExecutionListener実装例

```
@Component
public class ExitStatusChangeListener implements StepExecutionListener {

    @Override
    public ExitStatus afterStep(StepExecution stepExecution) {

        ExitStatus exitStatus = stepExecution.getExitStatus();
        if (conditionalCheck(stepExecution)) {
            // (1)
            exitStatus = new ExitStatus("CUSTOM STEP FAILED");
        }
        return exitStatus;
    }

    private boolean conditionalCheck(StepExecution stepExecution) {
        // omitted.
    }
}
```

#### ジョブ定義

```
<batch:step id="exitstatusjob.step">
    <batch:tasklet transaction-manager="transactionManager">
        <batch:chunk reader="reader" writer="writer" commit-interval="10" />
    </batch:tasklet>
    <batch:listeners>
        <batch:listener ref="exitStatusChangeListener"/>
    </batch:listeners>
</batch:step>
```

#### 実装内容の一覧

項番	説明
(1)	ステップの実行結果に応じて独自の終了コードを設定する。

- ジョブ終了時にステップが返却した終了状態を最終的なジョブの終了状態として反映する
  - `JobExecutionListener`の実装クラスで、`afterJob`メソッドに実装を行う。

### JobExecutionListener実装例

```
@Component
public class JobExitCodeChangeListener extends JobExecutionListenerSupport {

    @Override
    public void afterJob(JobExecution jobExecution) {
        // (1)
        if (jobExecution.getStepExecutions().stream()
            .anyMatch(s -> "CUSTOM STEP FAILED".equals(s.getExitStatus()
                .getExitCode())))) {
            jobExecution.setExitStatus(new ExitStatus("CUSTOM FAILED"));
        }
    }
}
```

### ジョブ定義

```
<batch:job id="exitstatusjob" job-repository="jobRepository">
    <batch:step id="exitstatusjob.step">
        <!-- omitted -->
    </batch:step>
    <batch:listeners>
        <batch:listener ref="jobExitCodeChangeListener"/>
    </batch:listeners>
</batch:job>
```

### 実装内容の一覧

項番	説明
(1)	ジョブの実行結果に応じて、最終的なジョブの終了コードを <code>JobExecution</code> に設定する。 ここではステップから返却された終了コードのいずれかに <code>CUSTOM STEP FAILED</code> が含まれている場合、終了コード <code>CUSTOM FAILED</code> としている。

#### 7.2.2.2. 終了コードのマッピングを追加定義する。

- ジョブの終了状態と終了コードのマッピングを定義する。

```

<!-- exitCodeMapper -->
<bean id="exitCodeMapper"
      class="org.springframework.batch.core.launch.support.SimpleJvmExitCodeMapper">
    <property name="mapping">
      <util:map id="exitCodeMapper" key-type="java.lang.String"
                 value-type="java.lang.Integer">
        <!-- ExitStatus -->
        <entry key="NOOP" value="0" />
        <entry key="COMPLETED" value="0" />
        <entry key="STOPPED" value="255" />
        <entry key="FAILED" value="255" />
        <entry key="UNKNOWN" value="255" />
        <entry key="CUSTOM FAILED" value="100" /> <!-- Custom Exit Status -->
      </util:map>
    </property>
  </bean>

```

プロセスの終了コードに1は厳禁



一般的にJavaプロセスはVMクラッシュやSIGKILLシグナル受信などによりプロセスが強制終了した際、終了ステータスとして1を返却することがある。正常・異常を問わずバッチアプリケーションの終了状態とは明確に区別すべきであるため、アプリケーション内ではプロセスの終了コードとして1を定義しないこと。

終了ステータスと終了コードの違いについて

[JobRepository](#)で管理されるジョブとステップの状態として、「ステータス([STATUS](#))」と「終了コード([EXIT\\_CODE](#))」があるが、以下の点で異なる。



- ステータスはSpring Batchの内部制御で用いられ enum型の[BatchStatus](#)による具体値が定義されているためカスタマイズできない。
- 終了コードはジョブのフロー制御やプロセス終了コードの変更で使用することができ、カスタマイズできる。

### 7.2.3. 二重起動防止

Spring Batchではジョブを起動する際、[JobRepository](#)から[JobInstance\(BATCH\\_JOB\\_INSTANCEテーブル\)](#)に對して以下の組み合わせが存在するか確認する。

- 起動対象となるジョブ名
- ジョブパラメータ

TERASOLUNA Batch 5.xではジョブ・ジョブパラメータの組み合わせが一致しても複数回起動可能としている。

つまり、二重起動を許容する。詳細は、[ジョブの起動パラメータ](#)を参照のこと。

二重起動を防止する場合は、ジョブスケジューラやアプリケーション内で実施する必要がある。

詳細な手段については、ジョブスケジューラ製品や業務要件に強く依存するため割愛する。  
個々のジョブについて、二重起動を抑止する必要があるかについて、検討すること。

## 7.2.4. ロギング

ログの設定方法について説明する。

ログの出力、設定、考慮事項はTERASOLUNA Server 5.xと共通点が多い。まずは、[ロギング](#)を参照のこと。

ここでは、TERASOLUNA Batch 5.x特有の考慮点について説明する。

### 7.2.4.1. ログ出力元の明確化

バッチ実行時のログは出力元のジョブやジョブ実行を明確に特定できるようにしておく必要がある。そのため、スレッド名、ジョブ名、実行ジョブIDを出力するとよい。特に非同期実行時は同名のジョブが異なるスレッドで並列に動作することになるため、ジョブ名のみの記録はログ出力元を特定しにくくなる恐れがある。

それぞれの要素は、以下の要領で実現できる。

スレッド名

`logback.xml`の出力パターンである`%thread`を指定する

ジョブ名・実行ジョブID

`JobExecutionListener`を実装したコンポーネントを作成し、ジョブの開始・終了時に記録する

## JobExecutionListener実装例

```
// package and import omitted.

@Component
public class JobExecutionLoggingListener implements JobExecutionListener {
    private static final Logger logger =
        LoggerFactory.getLogger(JobExecutionLoggingListener.class);

    @Override
    public void beforeJob(JobExecution jobExecution) {
        // (1)
        logger.info("job started. [JobName:{}][jobExecutionId:{}]",
            jobExecution.getJobInstance().getJobName(), jobExecution.getId());
    }

    @Override
    public void afterJob(JobExecution jobExecution) {
        // (2)
        logger.info("job finished.[JobName:{}][jobExecutionId:{}][ExitStatus:{}]"
            , jobExecution.getJobInstance().getJobName(),
            jobExecution.getId(), jobExecution.getExitStatus().getExitCode());
    }
}
```

## ジョブBean定義ファイル

```
<!-- omitted. -->
<batch:job id="loggingJob" job-repository="jobRepository">
    <batch:step id="loggingJob.step01">
        <batch:tasklet transaction-manager="jobTransactionManager">
            <!-- omitted. -->
        </batch:tasklet>
    </batch:step>
    <batch:listeners>
        <!-- (3) -->
        <batch:listener ref="jobExecutionLoggingListener"/>
    </batch:listeners>
</batch:job>
<!-- omitted. -->
```

## ジョブ名、ジョブ実行IDのログ出力実装例

項目番	説明
(1)	ジョブの開始前にジョブ名とジョブ実行IDをINFOログに出力している。
(2)	ジョブの終了時は(1)に加えて終了コードも出力している。

項目番	説明
(3)	コンポーネントとして登録されているJobExecutionLoggingListenerを特定ジョブのBean定義に関連づけている。

#### 7.2.4.2. ログ監視

バッチャーアプリケーションは運用時のユーザインターフェースはログが主体となる。監視対象と発生時のアクションを明確に設計しておかないと、フィルタリングが困難となり、対処に必要なログが埋もれてしまう危険がある。このため、ログの監視対象としてキーワードとなるメッセージやコード体系をあらかじめ決めておくとよい。ログに出力するメッセージ管理については、後述の[メッセージ管理](#)を参照。

#### 7.2.4.3. ログ出力先

バッチャーアプリケーションにおけるログの出力先について、どの単位でログを分散/集約するのかを設計するとよい。たとえばフラットファイルにログを出力する場合でも以下のように複数パターンが考えられる。

- 1ジョブあたり1ファイルに出力する
- 複数ジョブを1グループにまとめた単位で1ファイルに出力する
- 1サーバあたり1ファイルに出力する
- 複数サーバをまとめて1ファイルに出力する

いずれも対象システムにおける、ジョブ総数/ログ総量/発生するI/Oレートなどによって、どの単位でまとめるのが最適かが分かれる。また、ログを確認する方法にも依存する。ジョブスケジューラ上から参照することが多いか、コンソールから参照することが多いか、といった活用方法によっても選択肢が変わると想定する。

重要なことは、運用設計にてログ出力を十分検討し、試験にてログの有用性を確認することに尽きる。

### 7.2.5. メッセージ管理

メッセージ管理について説明する。

コード体系のばらつき防止や、監視対象のキーワードとしての抽出を設計しやすくするため、一定のルールに従ってメッセージを付与することが望ましい。

なお、ログと同様、メッセージ管理についても基本的にはTERASOLUNA Server 5.xと同様である。

## *MessageSource*の活用について

プロパティファイルからメッセージを使用するには*MessageSource*を使用することができます。

- 具体的な設定・実装例については[ログメッセージの一元管理](#)を参照のこと。
  - ここではログ出力のサンプルとしてSpring MVCのコントローラーのケースにそって例示されているが、Spring Batchの任意のコンポーネントに読み換えてほしい。
  - ここでは*MessageSource*のインスタンスを独自に生成しているが、TERASOLUNA Batch 5.xではその必要はない。*ApplicationContext*が生成された後でのみ、各コンポーネントにアクセスされるためである。なお、TERASOLUNA Batch 5.xでは以下のとおり設定済みである。



### *launch-context.xml*

```
<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource"
      p:basenames="classpath: META-INF/i18n/application-messages" />
```

## 7.3. Appendix. Spring Batch Admin

Spring Batch AdminはSpring Batchのサブプロジェクトであり、Webインターフェースによってジョブの実行状態を確認することができる。テスト/商用環境によらず手軽に参照できるため紹介する。

Spring Batch Adminはサンプルアプリケーションの形で配布されている。ここではWebコンテナとしてApache Tomcatを用い、warファイルをデプロイする。



Spring Batch Adminはジョブの実行状態・結果の確認だけでなく、ジョブの起動・停止も可能である。その場合は同様のwarファイルにジョブも同梱する必要があり、Webコンテナでジョブを実行することが必須という強い制約が生まれる。実行状態・結果の確認だけならばその必要性はないため、ここではあくまで参考方法として紹介する。

インストール手順は以下のとおり。

- [リリース配布サイト](#)より、1.3.1.RELEASEのzipをダウンロードし、任意の場所で展開する。
- 外部RDBMS定義のプロパティファイルbatch-RDBMSNAME.propertiesを作成する。
  - spring-batch-admin-1.3.1.RELEASE/spring-batch-admin-sample/src/main/resourcesに配置する。

### batch-postgresql.properties (PostgreSQLの場合)

```
# Placeholders batch.*  
#   for PostgreSQL:  
# (1)  
batch.jdbc.driver=org.postgresql.Driver  
batch.jdbc.url=jdbc:postgresql://localhost:5432/admin  
batch.jdbc.user=postgres  
batch.jdbc.password=postgres  
batch.jdbc.testWhileIdle=true  
batch.jdbc.validationQuery=SELECT 1  
# (2)  
batch.schema.script=classpath:/org/springframework/batch/core/schema-postgresql.sql  
batch.drop.script=classpath*:!/org/springframework/batch/core/schema-drop-  
postgresql.sql  
batch.business.schema.script=classpath:/business-schema-postgresql.sql  
batch.database.incrementer.class=org.springframework.jdbc.support.incrementer.PostgreSQLSequenceMaxValueIncrementer  
  
# Non-platform dependent settings that you might like to change  
# (3)  
batch.data.source.init=false
```

### 設定内容の項目一覧(PostgreSQL)

項目番	説明
(1)	接続先RDBMSのJDBCドライバ設定を記述する。
(2)	JobRepository及び業務データベース初期化時のスクリプトを記述する。(3)の理由により使用しないが、batch.xxxxが不足した場合起動時にエラーとなるため、ダミーでもよいので記載する。
(3)	Spring Batch Admin起動時に JobRepository及び業務データベーススキーマを初期化しないよう必ずfalseを明示する。 この設定が記載されていない場合、RDBMS上の管理・業務データソースがすべてクリアされてしまう。

- spring-batch-admin-1.3.1.RELEASE/spring-batch-admin-sample/配下のpom.xmlに、 JDBCドライバの依存ライブラリを追加する。

```
<project>
  <!-- omitted. -->
  <dependencies>
    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>9.4.1212.jre7</version>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
  <!-- omitted. -->
</project>
```

- mvn clean packageコマンドによりwarファイルを作成する。
- 環境変数JAVA\_OPTSに外部RDBMS名-DENVIRONMENT=postgresqlを設定し、Tomcatを起動する。

#### 環境変数の設定とTomcat起動

```
$ export JAVA_OPTS="$JAVA_OPTS -DENVIRONMENT=postgresql"
$ echo $JAVA_OPTS
-DENVIRONMENT=postgresql
$ TOMCAT_HOME/bin/catalina.sh run
```

- target/spring-batch-admin-1.3.1.warをTomcatにデプロイする。
- ブラウザで<http://tomcathost:port/spring-batch-admin-sample-1.3.1.RELEASE/>を開き、Jobsを選択する。

Resource	Method	Description
/configuration	GET	Form for upload of Job configuration file
/files	GET	List uploaded data files
/files	POST	Upload a new file (in a multipart form request) with path= and file= parameters.
/files	DELETE	Remove a file that was previously uploaded. Add pattern= parameter to narrow down to a subset of existing resources.
/files/{path}	GET	Download a previously uploaded file. The path can contain directory separators.
/home	GET	List the resources available

### Spring Batch Admin ルート画面

- 実行状態・実行結果取得対象のジョブ名を選択する。

Name	Description	Execution Count	Launchable	Incrementable
job2	No description	0	true	false
job1	No description	0	true	true
infinite	No description	0	true	false
job01	No description	2	false	false
jobStopAndRestartTask	No description	2	false	false

### Spring Batch Admin ジョブ選択画面

- 対象ジョブの実行状態・結果が表示される。

The screenshot shows a web browser window titled "Spring Batch Admin: Job Summary" with the URL "localhost:8080/spring-batch-admin-sample-1.3.1.RELEASE/jobs/job01". The page header includes the Spring Source logo and navigation links for Home, Jobs, Executions, Files, Spring, and Spring Batch.

### Job Instances for Job (job01)

ID	JobExecution Count	Last JobExecution	Last JobExecution Date	Last JobExecution Start	Last JobExecution Duration	Last JobExecution Parameters	
6	<a href="#">executions</a>	1	COMPLETED	2017-02-13	14:06:27	00:00:00	{jsr_batch_run_id=param1=value1}
4	<a href="#">executions</a>	1	COMPLETED	2017-02-13	14:05:16	00:00:00	{jsr_batch_run_id=param1=value1}

Rows: 1-2 of 2 Page Size: 20

The table above shows instances of this job with an indication of the status of the last execution. If you want to look at all executions for [see here](#).

Spring Batch Admin ジョブの状態・結果画面

# Chapter 8. フロー制御と並列・多重処理

## 8.1. フロー制御

### 8.1.1. Overview

1つの業務処理を実装する方法として、1つのジョブに集約して実装するのではなく、複数のジョブに分割し組み合わせることで実装することがある。このとき、ジョブ間の依存関係を定義したものをジョブネットと呼ぶ。

ジョブネットを定義することのメリットを下記に挙げる。

- 処理の進行状況が可視化しやすくなる
- ジョブの部分再実行、実行保留、実行中止が可能になる
- ジョブの並列実行が容易になる

以上より、バッチ処理を設計する場合はジョブネットも併せてジョブ設計を行うことが一般的である。

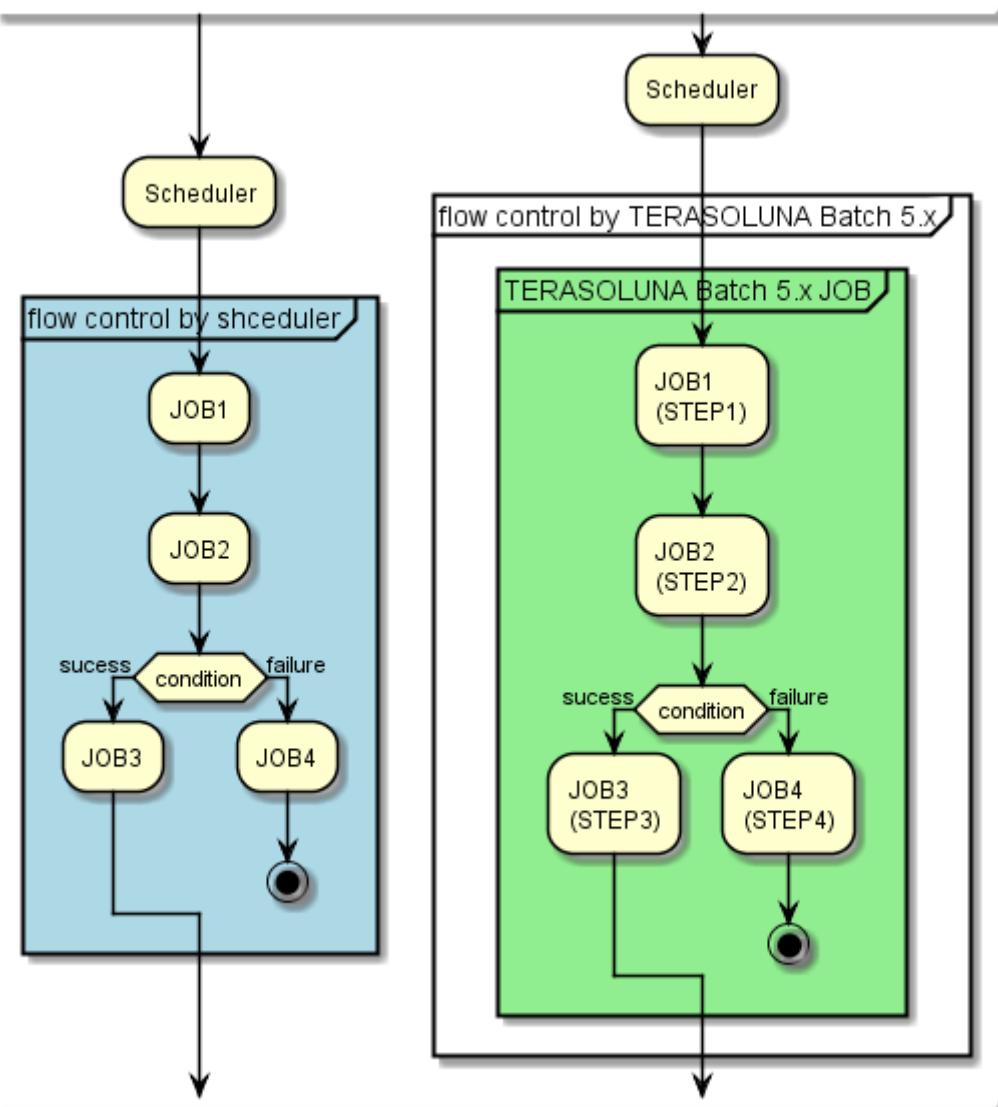
#### 処理内容とジョブネットの適性



分割するまでもないシンプルな業務処理やオンライン処理と連携する処理に対して、ジョブネットは適さないことが多い。

本ガイドラインでは、ジョブネットでジョブ同士の流れを制御することをフロー制御と呼ぶ。また処理の流れにおける前のジョブを先行ジョブ、後のジョブを後続ジョブと呼び、先行ジョブと後続ジョブの依存関係を、先行後続関係と呼ぶ。

フロー制御の概念図を以下に示す。



フロー制御の概念図

上図のとおり、フロー制御はジョブスケジューラ、TERASOLUNA Batch 5.xのどちらでも実施可能である。しかし、以下の理由によりできる限りジョブスケジューラを活用することが望ましい。

#### TERASOLUNA Batch 5.xで実現した場合

- 1ジョブの処理や状態が多岐に渡る傾向が強まり、ブラックボックス化しやすい。
- ジョブスケジューラとジョブの境界があいまいになってしまう
- ジョブスケジューラ上から異常時の状況がみえにくくなってしまう

ただし、ジョブスケジューラに定義するジョブ数が多くなると、以下の様なデメリットが生じることも一般に知られている。

- ジョブスケジューラによる以下のようなコストが累積し、システム全体の処理時間が伸びる
  - ジョブスケジューラ製品固有の通信、実行ノードの制御、など
  - ジョブごとのJavaプロセス起動に伴うオーバーヘッド
- ジョブ登録数の限界

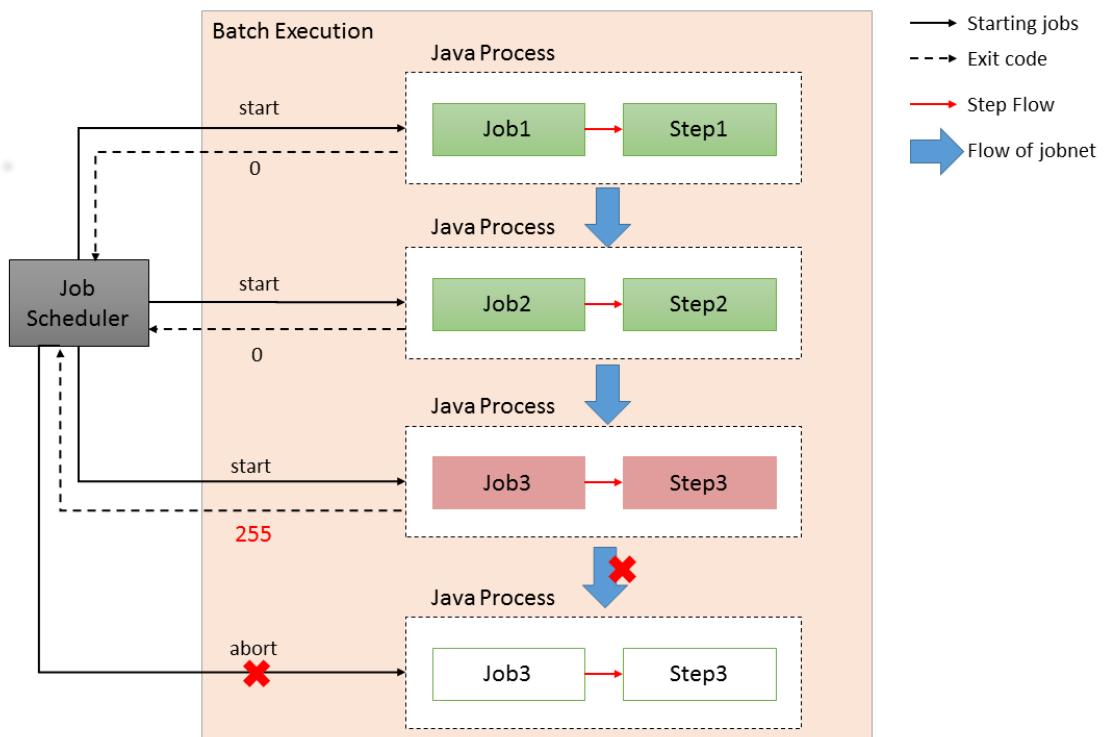
このため、以下を方針とする。

- 基本的にはジョブスケジューラによりフロー制御を行う。
- ジョブ数が多いことによる弊害がある場合に限り、以下のとおり対処する。
  - TERASOLUNA Batch 5.xにてシーケンシャルな複数の処理を1ジョブにまとめる。
    - シンプルな先行後続関係を1ジョブに集約するのみとする。
    - ステップ終了コードの変更と、この終了コードに基づく後続ステップ起動の条件分岐は機能上利用可能だが、ジョブの実行管理が複雑化するため、ジョブ終了時のプロセス終了コード決定に限り原則利用する。
  - どうしても条件分岐を使わないと問題を解消できない場合に限り使用を許容するが、シンプルな先行後続関係を維持するよう配慮すること。



ジョブの終了コードの決定について、詳細は終了コードのカスタマイズを参照。

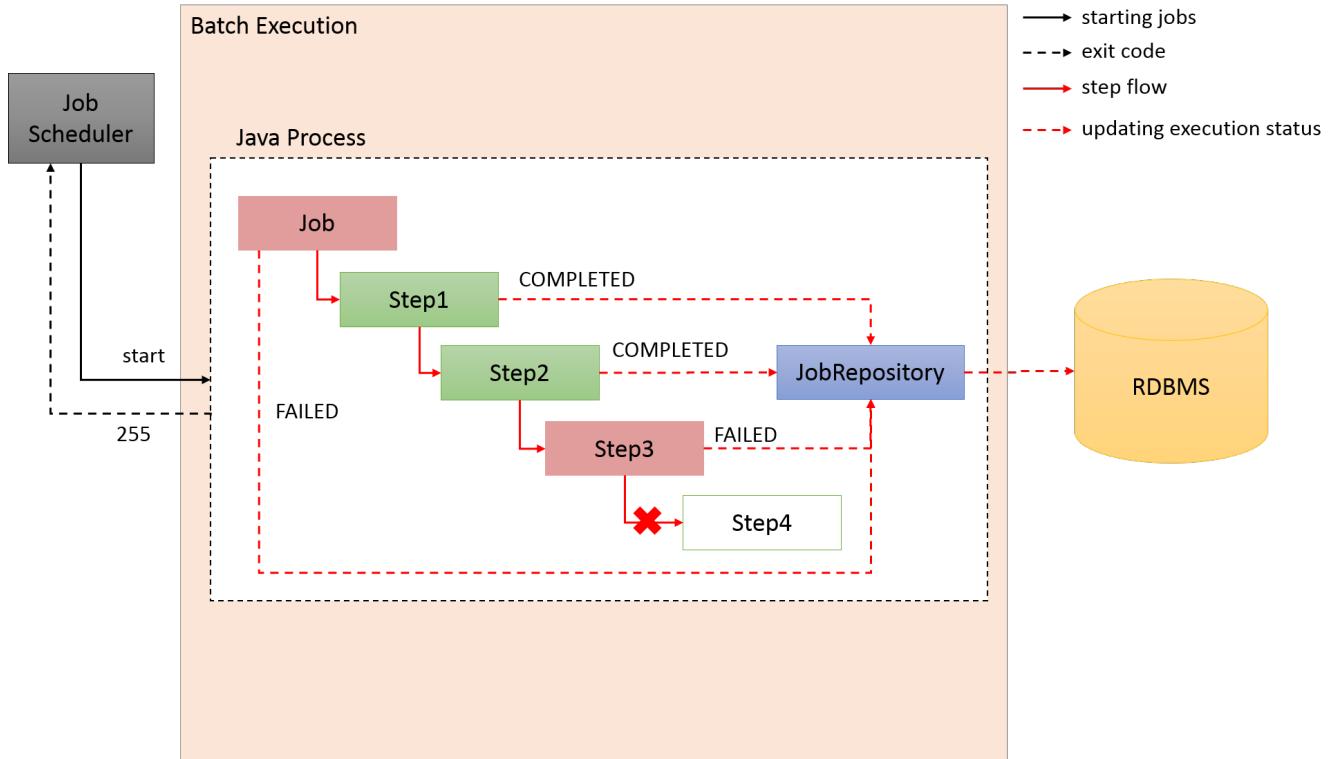
また、以下に先行後続を実現する上で意識すべきポイントを示す。



ジョブスケジューラによるフロー制御

#### 意識すべきポイント

- ジョブスケジューラがシェル等を介してjavaプロセスを起動する。
- 1ジョブが1javaプロセスとなる。
  - 処理全体では、4つのjavaプロセスが起動する。
- ジョブスケジューラが各処理の起動順序を制御する。それぞれのjavaプロセスは独立している。
- 後続ジョブの起動判定として、先行ジョブのプロセス終了コードが用いられる。
- ジョブ間のデータ受け渡しは、ファイルやDBなど外部リソースを使用する必要がある。



### TERASOLUNA Batch 5.xによるフロー制御

#### 意識すべきポイント

- ジョブスケジューラがシェル等を介してjavaプロセスを起動する。
- 1ジョブが1javaプロセスとなる。
  - 処理全体では、1つのjavaプロセスしか使わない。
- 1javaプロセス内で各ステップの起動順序を制御する。それぞれのステップは独立している。
- 後続ステップの起動判定として、先行ステップの終了コードが用いられる。
- ステップ間のデータはインメモリで受け渡しが可能である。

以降、TERASOLUNA Batch 5.xによるフロー制御の実現方法について説明する。

ジョブスケジューラでのフロー制御は製品仕様に強く依存するためここでは割愛する。

#### フロー制御の応用例



複数ジョブの並列化・多重化は、一般的にジョブスケジューラとジョブネットによって実現することが多い。

しかし、TERASOLUNA Batch 5.xではフロー制御の機能を応用し、複数ジョブの並列化、多重化を実現する方法を説明している。詳細は、[並列処理と多重処理](#)を参照。

本機能は、チャンクモデルとタスクレットモデルとで同じ使い方になる。

#### 8.1.2. How to use

TERASOLUNA Batch 5.xでのフロー制御方法を説明する。

### 8.1.2.1. シーケンシャルフロー

シーケンシャルフローとは先行ステップと後続ステップを直列に連結したフローである。何らかの業務処理がシーケンシャルフロー内のステップで異常終了した場合、後続ステップは実行されずにジョブが中断する。このとき、**JobRepository**によりジョブ実行IDに紐付けられる当該のステップとジョブのステータス・終了コードは**FAILED**として記録される。失敗原因の回復後にリスタートを実施することで、異常終了したステップから処理をやり直すことができる。



ジョブのリスタート方法については[ジョブのリスタート](#)を参照。

ここでは3つのステップからなるジョブのシーケンシャルフローを設定する。

#### Bean定義

```
<bean id="sequentialFlowTasklet"
      class="org.terasoluna.batch.functionaltest.ch08.flowcontrol.SequentialFlowTasklet"
      p:failExecutionStep="#{jobParameters[failExecutionStep]}" scope="step"/>

<batch:step id="parentStep">
    <batch:tasklet ref="sequentialFlowTasklet"
                    transaction-manager="jobTransactionManager"/>
</batch:step>

<batch:job id="jobSequentialFlow" job-repository="jobRepository">
    <batch:step id="jobSequentialFlow.step1"
                next="jobSequentialFlow.step2" parent="parentStep"/> <!-- (1) -->
    <batch:step id="jobSequentialFlow.step2"
                next="jobSequentialFlow.step3" parent="parentStep"/> <!-- (1) -->
    <batch:step id="jobSequentialFlow.step3" parent="parentStep"/> <!-- (2) -->
</batch:job>
```

項目番号	説明
(1)	<batch:step>で、このステップの正常終了後に起動する後続ステップを指定する。 next属性に後続ステップのidを設定する。
(2)	フローの末端になるステップには、next属性は不要となる。

これにより、以下の順でステップが直列に起動する。

jobSequentialFlow.step1 → jobSequentialFlow.step2 → jobSequentialFlow.step3

<batch:flow>を使った定義方法

前述の例では<batch:job>内に直接フローを定義した。<batch:flow>を利用して、フロー定義を外部に切り出すこともできる。以下に<batch:flow>を利用した場合の例を示す。

```
<batch:job id="jobSequentialOuterFlow" job-repository="jobRepository">
    <batch:flow id="innerFlow" parent="outerFlow"/> <!-- (1) -->
</batch:job>

<!-- (2) -->
<batch:flow id="outerFlow">
    <batch:step id="jobSequentialOuterFlow.step1"
        next="jobSequentialOuterFlow.step2"
        parent="parentStep"/>
    <batch:step id="jobSequentialOuterFlow.step2"
        next="jobSequentialOuterFlow.step3"
        parent="parentStep"/>
    <batch:step id="jobSequentialOuterFlow.step3"
        parent="parentStep"/>
</batch:flow>
```



項目番号	説明
(1)	parent属性に(2)で定義したフローのidを設定する。
(2)	シーケンシャルフローを定義する。

### 8.1.2.2. ステップ間のデータの受け渡し

Spring Batchには、ステップ、ジョブそれぞれのスコープで利用できる実行コンテキストのExecutionContextが用意されている。実行コンテキストを利用することでステップ内のコンポーネント間でデータを共有できる。このとき、ステップの実行コンテキストはステップ間で共有できないため、先行ステップの実行コンテキストは後続ステップからは参照できない。ジョブの実行コンテキストを利用すれば実現可能だが、すべてのステップから参照可能になるため、慎重に扱う必要がある。ステップ間の情報を引き継ぐ必要があるときは、以下の手順により対応できる。

1. 先行ステップの後処理で、ステップスコープの実行コンテキストに格納した情報をジョブスコープの実行コンテキストに移す。
2. 後続ステップがジョブスコープの実行コンテキストから情報を取得する。

最初の手順は、Spring Batchから提供されているExecutionContextPromotionListenerを利用することで、実装をせずとも、引き継ぎたい情報をリスナーに指定するだけ実現できる。

*ExecutionContext*を使用する上での注意点

データの受け渡しに使用する*ExecutionContext*は*JobRepository*によりRDBMSの*BATCH\_JOB\_EXECUTION\_CONTEXT*、*BATCH\_JOB\_STEP\_EXECUTION\_CONTEXT*にシリアル化された状態で保存されるため、以下3点に注意すること。

1. 受け渡しデータはシリアル化可能な形式のオブジェクトであること。
  - `java.io.Serializable`を実装している必要がある。
2. 受け渡しデータは必要最小限に留めること。

*ExecutionContext*はSpring Batchによる実行制御情報の保存でも利用しており、受け渡しデータが大きくなればそれだけシリアル化コストが増大する。
3. データ受け渡しに直接ジョブ実行コンテキストに保存せず、上述の*ExecutionContextPromotionListener*を使用すること。

ジョブ実行コンテキストはステップよりスコープが広いため、無用なシリアル化データが蓄積しやすいため。



また、実行コンテキストを経由せず、*Singleton*や*Job*スコープのBeanを共有することでも情報のやり取りは可能だが、この方法もサイズが大きすぎるとメモリリソースを圧迫する可能性があるので注意すること。

以下、タスクレットモデルとチャンクモデルについて、それぞれステップ間のデータ受け渡しについて説明する。

#### 8.1.2.2.1. タスクレットモデルを用いたステップ間のデータ受け渡し

受け渡しデータの保存・取得に、*ChunkContext*から*ExecutionContext*を取得し、ステップ間のデータ受け渡しを行う。

## データ受け渡し元タスクレットの実装例

```
// package, imports are omitted.

@Component
public class SavePromotionalTasklet implements Tasklet {

    // omitted.

    @Override
    public RepeatStatus execute(StepContribution contribution,
        ChunkContext chunkContext) throws Exception {

        // (1)
        chunkContext.getStepContext().getStepExecution().getExecutionContext()
            .put("promotion", "value1");

        // omitted.

        return RepeatStatus.FINISHED;
    }
}
```

## データ受け渡し先のタスクレット実装例

```
// package and imports are omitted.

@Component
public class ConfirmPromotionalTasklet implements Tasklet {

    @Override
    public RepeatStatus execute(StepContribution contribution,
        ChunkContext chunkContext) {
        // (2)
        Object promotion = chunkContext.getStepContext().getJobExecutionContext()
            .get("promotion");

        // omitted.

        return RepeatStatus.FINISHED;
    }
}
```

## ジョブBean定義の記述例

```
<!-- import,annotation,component-scan definitions are omitted -->

<batch:job id="jobPromotionalFlow" job-repository="jobRepository">
    <batch:step id="jobPromotionalFlow.step1" next="jobPromotionalFlow.step2">
        <batch:tasklet ref="savePromotionalTasklet"
                        transaction-manager="jobTransactionManager"/>
        <batch:listeners>
            <batch:listener>
                <!-- (3) -->
                <bean
                    class="org.springframework.batch.core.listener.ExecutionContextPromotionListener"
                    p:keys="promotion"
                    p:strict="true"/>
            </batch:listener>
        </batch:listeners>
    </batch:step>
    <batch:step id="jobPromotionalFlow.step2">
        <batch:tasklet ref="confirmPromotionalTasklet"
                        transaction-manager="jobTransactionManager"/>
    </batch:step>
</batch:job>
<!-- omitted -->
```

### 実装内容の説明

項番	説明
(1)	ステップ実行コンテキストのExecutionContextに後続ステップに受け渡す値を設定する。ここでは一連のデータ受け渡しに必要なキーとして、promotionを指定している。
(2)	先行ステップの(1)で設定された受け渡しデータをExecutionContextから、受け渡し元で指定されたキーpromotionを用いて取得する。 ここで使用しているExecutionContextは(1)のステップ実行コンテキストではなく、ジョブ実行コンテキストである点に注意する。
(3)	ExecutionContextPromotionListenerを用い、 ステップ実行コンテキストからジョブ実行コンテキストに受け渡しデータを移す。 keysプロパティには(1)で指定した受け渡しキーを指定する。 strict=trueプロパティにより、ステップ実行コンテキストに存在しない場合はIllegalArgumentExceptionがスローされる。falseの場合は受け渡しデータがなくても処理が継続する。

ExecutionContextPromotionListenerとステップ終了コードについて



ExecutionContextPromotionListenerはデータ受け渡し元のステップ終了コードが正常終了時(COMPLETED)の場合のみ、ステップ実行コンテキストからジョブ実行コンテキストへデータを移す。  
後続ステップが継続して実行される終了コードのカスタマイズを行う場合、statusプロパティに終了コードを配列形式で指定すること。

#### 8.1.2.2.2. チャンクモデルを用いたステップ間のデータ受け渡し

`ItemProcessor`に`@AfterStep`、`@BeforeStep`アノテーションを付与したメソッドを使用する。データ受け渡しに使用するリスナーと、`ExecutionContext`の使用方法はタスクレットと同様である。

##### データ受け渡し元`ItemProcessor`の実装例

```
// package and imports are omitted.

@Component
@Scope("step")
public class PromotionSourceItemProcessor implements ItemProcessor<String, String> {

    @Override
    public String process(String item) {
        // omitted.
    }

    @AfterStep
    public ExitStatus afterStep(StepExecution stepExecution) {
        // (1)
        ExecutionContext jobContext = stepExecution.getExecutionContext();
        // (2)
        jobContext.put("promotion", "value2");
        return null;
    }
}
```

##### データ受け渡し先`ItemProcessor`の実装例

```
// package and imports are omitted.

@Component
@Scope("step")
public class PromotionTargetItemProcessor implements ItemProcessor<String, String> {

    @Override
    public String process(String item) {
        // omitted.
    }

    @BeforeStep
    public void beforeStep(StepExecution stepExecution) {
        // (3)
        ExecutionContext jobContext = stepExecution.getJobExecution()
            .getExecutionContext();
        // omitted.
    }
}
```

## ジョブBean定義の記述例

```
<!-- import,annotation,component-scan definitions are omitted -->
<batch:job id="jobChunkPromotionalFlow" job-repository="jobRepository">
    <batch:step id="jobChunkPromotionalFlow.step1" parent="sourceStep"
        next="jobChunkPromotionalFlow.step2">
        <batch:listeners>
            <batch:listener>
                <!-- (4) -->
                <bean
                    class="org.springframework.batch.core.listener.ExecutionContextPromotionListener"
                    p:keys="promotion"
                    p:strict="true" />
            </batch:listener>
        </batch:listeners>
    </batch:step>
    <batch:step id="jobChunkPromotionalFlow.step2" parent="targetStep"/>
</batch:job>

<!-- step definitions are omitted. -->
```

## 実装内容の説明

項目番	説明
(1)	ステップ実行コンテキストのExecutionContextに後続ステップに受け渡す値を設定する。ここでは一連のデータ受け渡しに必要なキーとして、promotionを指定している。
(2)	先行ステップの(1)で設定された受け渡しデータをExecutionContextから、受け渡し元で指定されたキーpromotionを用いて取得する。ここで使用しているExecutionContextは(1)のステップ実行コンテキストではなく、ジョブ実行コンテキストである点に注意する。
(3)	ExecutionContextPromotionListenerを用い、ステップ実行コンテキストからジョブ実行コンテキストに受け渡しデータを移す。プロパティの指定はタスクレットと同様である。

### 8.1.3. How to extend

ここでは後続ステップの条件分岐と、条件により後続ステップ実行前にジョブを停止させる停止条件について説明する。

ジョブ・ステップの終了コードとステータスの違い。



以降の説明では「ステータス」と「終了コード」という言葉が頻繁に登場する。これらの判別がつかない場合混乱を招く恐れがあるため、まず終了コードのカスタマイズを参照してほしい。

#### 8.1.3.1. 条件分岐

条件分岐は先行ステップの実行結果となる終了コードを受けて、複数の後続ステップから1つを選択して継続実行させることを言う。

いずれの後続ステップを実行させずにジョブを停止させる場合は後述の[停止条件](#)を参照。

### ジョブBean定義記述例

```
<batch:job id="jobConditionalFlow" job-repository="jobRepository">
    <batch:step id="jobConditionalFlow.stepA" parent="conditionalFlow.parentStep">
        <!-- (1) -->
        <batch:next on="COMPLETED" to="jobConditionalFlow.stepB" />
        <batch:next on="FAILED" to="jobConditionalFlow.stepC"/>
    </batch:step>
    <!-- (2) -->
    <batch:step id="jobConditionalFlow.stepB" parent="conditionalFlow.parentStep"/>
    <!-- (3) -->
    <batch:step id="jobConditionalFlow.stepC" parent="conditionalFlow.parentStep"/>
</batch:job>
```

### 実装内容の説明

項番	説明
(1)	シーケンシャルフローのように <code>&lt;batch:step&gt;</code> 要素内に <code>next</code> 属性を指定せず、 <code>&lt;batch:next&gt;</code> を複数置くことで、 <code>to</code> 属性で指定される後続ステップに振り分けることができる。 onには遷移条件となるステップの終了コードを指定する。
(2)	(1)のステップ終了コードが <code>COMPLETED</code> の場合のみに実行される後続ステップとなる。
(3)	(1)のステップ終了コードが <code>FAILED</code> の場合のみに実行される後続ステップとなる。 この指定が行われることで先行ステップ処理失敗時にジョブが停止せず、回復処理などの後続ステップが実行される。

#### 後続ステップによる回復処理の注意点

先行ステップの処理失敗(終了コードが`FAILED`)により後続ステップの回復処理が行われた場合、回復処理の成否を問わず先行ステップのステータスは`ABANDONED`となり、リスタート不能となる。



後続ステップの回復処理が失敗した場合にジョブをリスタートすると、回復処理のみが再実行される。

このため、先行ステップを含めて処理をやり直す場合は別のジョブ実行としてリランさせる必要がある。

#### 8.1.3.2. 停止条件

先行ステップの終了コードに応じ、ジョブを停止させる方法を説明する。

停止の手段として、以下の3つの要素を指定する方法がある。

1. `end`
2. `fail`
3. `stop`

これらの終了コードが先行ステップに該当する場合は後続ステップが実行されない。

また、同一ステップ内にそれぞれ複数指定が可能である。

## ジョブBean定義記述例

```
<batch:job id="jobStopFlow" job-repository="jobRepository">
    <batch:step id="jobStopFlow.step1" parent="stopFlow.parentStep">
        <!-- (1) -->
        <batch:end on="END_WITH_NO_EXIT_CODE"/>
        <batch:end on="END_WITH_EXIT_CODE" exit-code="COMPLETED_CUSTOM"/>
        <!-- (2) -->
        <batch:next on="*" to="jobStopFlow.step2"/>
    </batch:step>
    <batch:step id="jobStopFlow.step2" parent="stopFlow.parentStep">
        <!-- (3) -->
        <batch:fail on="FORCE_FAIL_WITH_NO_EXIT_CODE"/>
        <batch:fail on="FORCE_FAIL_WITH_EXIT_CODE" exit-code="FAILED_CUSTOM"/>
        <!-- (2) -->
        <batch:next on="*" to="jobStopFlow.step3"/>
    </batch:step>
    <batch:step id="jobStopFlow.step3" parent="stopFlow.parentStep">
        <!-- (4) -->
        <batch:stop on="FORCE_STOP" restart="jobStopFlow.step4" exit-code="" />
        <!-- (2) -->
        <batch:next on="*" to="jobStopFlow.step4"/>
    </batch:step>
    <batch:step id="jobStopFlow.step4" parent="stopFlow.parentStep"/>
</batch:job>
```

## ジョブの停止の設定内容説明

項番	説明
(1)	<batch:end>のon属性とステップ終了コードが一致した場合、ジョブは正常終了(ステータス : COMPLETED)としてJobRepositoryに記録される。 exit-code属性を付与した場合、ジョブの終了コードをデフォルトのCOMPLETEDからカスタマイズすることができる。
(2)	<batch:next>のon属性にワイルドカード(*)を指定することで、end、fail、stopいずれの終了コードにも該当しない場合に後続ジョブを継続させることができる。 ここではステップ要素内の最後に記述しているが、終了コードの一致条件が先に評価されるため、要素の並び順はステップ要素内であれば任意である。
(3)	<batch:fail>を使用した場合、ジョブは異常終了(ステータス : FAILED)としてJobRepositoryに記録される。 <batch:end>と同様、exit-code属性を付与することで、ジョブの終了コードをデフォルトのFAILEDからカスタマイズすることができる。
(4)	<batch:stop>を使用した場合、ステップの正常終了時にジョブは中断(ステータス : STOPPED)としてJobRepositoryに記録される。 restart属性はリストアート時に中断状態からジョブが再開されるステップを指定する。 <batch:end>と同様、exit-code属性を付与することができるが、空白文字列を指定すること。(後述のコラムを参照)



`exit-code`属性による終了コードのカスタマイズ時は漏れなくプロセス終了コードにマッピングさせること。

詳細は[終了コードのカスタマイズ](#)を参照。

`<batch:stop>`で`exit-code`に空文字列を指定すること。

```
<step id="step1" parent="s1">
    <stop on="COMPLETED" restart="step2"/>
</step>

<step id="step2" parent="s2"/>
```



上記は`step1`が正常終了した際ジョブは停止状態となり、再度リスタート実行時に`step2`を実行させることを意図したフロー制御になっている。

しかし、Spring Batchの不具合により、現在意図したとおりに動作しない。リスタート後に`step2`が実行されることはなく、ジョブの終了コードは`NOOP`となり、ステータスが`COMPLETED`となる。

これを回避するためには上述で示したように`exit-code`で""(空文字)を付与すること。

不具合の詳細は[Spring Batch/BATCH-2315](#)を参照。

## 8.2. 並列処理と多重処理

### 8.2.1. Overview

一般的に、バッチウィンドウ(バッチ処理のために使用できる時間)がシビアなバッチシステムでは、複数ジョブを並列に動作させる(以降、並列処理と呼ぶ)ことで全体の処理時間を可能な限り短くするように設計する。

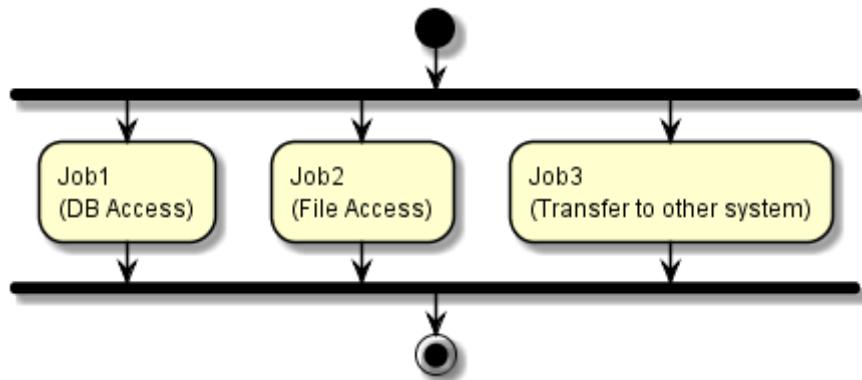
しかし、1ジョブの処理データが大量であるために処理時間がバッチウィンドウに収まらない場合がある。

その際は、1ジョブの処理データを分割して多重走行させる(以降、多重処理と呼ぶ)ことで処理時間を短縮させる手法が用いられる。

この、並列処理と多重処理は同じような意味合いで扱われることもあるが、ここでは以下の定義とする。

#### 並列処理

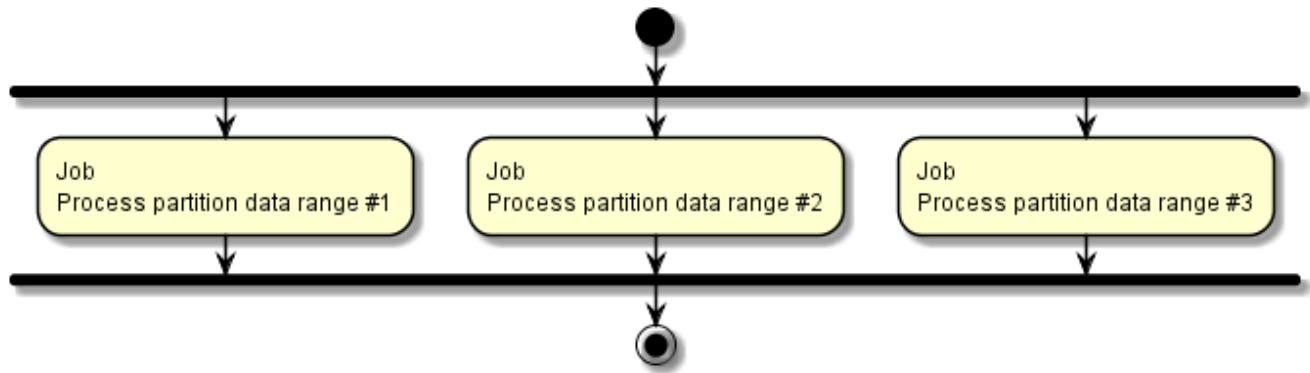
複数の異なるジョブを、同時に実行する。



並列処理の概略図

#### 多重処理

1ジョブの処理対象を分割して、同時に実行する。



多重処理の概略図

並列処理と多重処理ともにジョブスケジューラで行う方法とTERASOLUNA Batch 5.xで行う方法がある。なお、TERASOLUNA Batch 5.xでの並列処理および多重処理は [フロー制御](#)の上に成り立っている。

#### 並列処理および多重処理の実現方法

実現方法	並列処理	多重処理
ジョブスケジューラ	依存関係がない複数の異なるジョブを同時に実行するように定義する。	複数の同じジョブを異なるデータ範囲で実行するように定義する。各ジョブに処理対象のデータを絞るための情報を引数などで渡す。 たとえば、1年間のデータを月ごとに分割する、エリアや支店などの単位で分割する、など
TERASOLUNA Batch 5.x	<p><a href="#">Parallel Step (並列処理)</a>          ステップ単位で並列処理を行う。          各ステップは同じ処理である必要はなく、DBとファイルというような種類が異なるリソースに対して並列で処理を行う事も可能である。</p>	<p><a href="#">Partitioning Step (多重処理)</a>          マスタステップでは対象データを分割するためのキーを取得し、スレーブステップではこのキーにもとづいて分割したデータを処理する。  <b>Parallel Step</b>とは異なりスレーブステップの処理は同一処理となる。</p>

### ジョブスケジューラを使用する場合

1ジョブに1プロセスが割り当てられるため複数プロセスで起動される。そのため、1つのジョブを設計・実装する難易度は低い。  
 しかし、複数プロセスで起動するため、同時実行数が増えるとマシンリソースへの負荷が高くなる。よって、同時実行数が3、4程度であれば、ジョブスケジューラを利用するとよい。  
 もちろん、この数値は絶対的なものではない。実行環境やジョブの実装に依存するため目安としてほしい。

### TERASOLUNA Batch 5.xを使用する場合

各ステップがスレッドに割り当てられるため、1プロセス複数スレッドで動作する。そのため、1つのジョブへの設計・実装の難易度はジョブスケジューラを使用する場合より高くなる。  
 しかし、複数スレッドで起動するため、同時実行数が増えてもマシンリソースへの負荷がジョブスケジューラを使用する場合ほど高くはならない。よって、同時実行数が多い(5以上の)場合であれば、TERASOLUNA Batch 5.xを利用するのがよい。  
 もちろん、この数値は絶対的なものではない。実行環境やシステム特性に依存するため目安としてほしい。

Spring Batchで実行可能な並列処理方法の1つにMulti Thread Stepがあるが、以下の理由によりTERASOLUNA Batch 5.xでの利用は非推奨とする。

#### Multi Thread Stepとは

チャンク単位で複数スレッドで並列処理を行う方法。

#### 非推奨理由

Spring Batchが提供しているReaderやWriterのほとんどが、マルチスレッドでの利用を想定して設計されていない。そのため、データロストや重複処理が発生する可能性があり、処理の信頼性が低い。また、複数スレッドで動作するため、一定の処理順序とならない。

ItemReader/ItemProcessor/ItemWriterを自作する場合でもスレッドセーフなどMulti Thread Stepを使うためには考慮すべき点が多く実装および運用の難易度が高くなる。これらの理由により、Multi Thread Stepは非推奨としている。代わりにPartitioning Step (多重処理)を利用することを推奨する。

 `org.springframework.batch.item.support.SynchronizedItemStreamReader`を利用することで、既存のItemReaderをスレッドセーフにすることは可能である。それでも処理順序の課題は残るため、TERASOLUNA Batch 5.xではMulti Thread Stepは利用しないこと。

 並列処理・多重処理で1つのデータベースに対して更新する場合は、リソース競合とデッドロックが発生する可能性がある。ジョブ設計の段階から潜在的な競合発生を排除すること。

マルチプロセスや複数筐体への分散処理は、Spring Batchに機能があるが、TERASOLUNA Batch 5.xとしては障害設計が困難になるため扱わないこととする。

本機能は、チャンクモデルとタスクレットモデルとで同じ使い方になる

#### 8.2.1.1. ジョブスケジューラによる並列処理と多重処理

ここでは、ジョブスケジューラによる並列処理と多重処理の概念について説明を行う。

ジョブ登録、スケジュール設定などについては、使用するジョブスケジューラのマニュアルを参照してほしい。

##### 8.2.1.1.1. ジョブスケジューラによるジョブの並列化

並列実行させたい処理をそれぞれジョブとして登録、それぞれのジョブが同時に開始するようにスケジュールを設定する。各々のジョブは異なる処理として登録してよい。

##### 8.2.1.1.2. ジョブスケジューラによるジョブの多重化

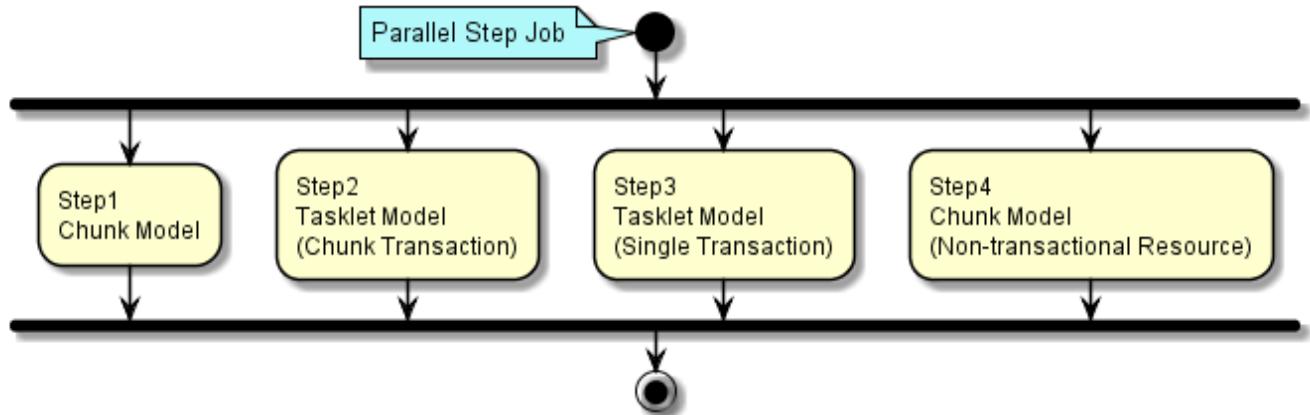
多重実行させたい処理を複数登録し、パラメータにより対象データの抽出範囲を指定する。その上で、それぞれのジョブが同時に開始するようにスケジュールを設定する。各々のジョブは同じ処理ではあるが、処理対象データ範囲は独立していることが必要となる。

## 8.2.2. How to use

TERASOLUNA Batch 5.xでの並列処理および多重処理を行う方法を説明する。

### 8.2.2.1. Parallel Step (並列処理)

Parallel Step (並列処理)の方法を説明する。



Parallel Stepの概要図

#### 概要図の説明

各ステップに別々な処理を定義することができ、それらを並列に実行することができる。各ステップごとにスレッドが割り当てられる。

Parallel Stepの概要図を例にしたParallel Stepの定義方法を以下に示す。

## Parallel Stepのジョブ定義

```
<!-- Task Executor -->
<!-- (1) -->
<task:executor id="parallelTaskExecutor" pool-size="10" queue-capacity="200"/>

<!-- Job Definition -->
<!-- (2) -->
<batch:job id="parallelStepJob" job-repository="jobRepository">
    <batch:split id="parallelStepJob.split" task-executor="parallelTaskExecutor">
        <batch:flow> <!-- (3) -->
            <batch:step id="parallelStepJob.step.chunk.db">
                <!-- (4) -->
                <batch:tasklet transaction-manager="jobTransactionManager">
                    <batch:chunk reader="fileReader" writer="databaseWriter"
                        commit-interval="100"/>
                </batch:tasklet>
            </batch:step>
        </batch:flow>

        <batch:flow> <!-- (3) -->
            <batch:step id="parallelStepJob.step.tasklet.chunk">
                <!-- (5) -->
                <batch:tasklet transaction-manager="jobTransactionManager"
                    ref="chunkTransactionTasklet"/>
            </batch:step>
        </batch:flow>

        <batch:flow> <!-- (3) -->
            <batch:step id="parallelStepJob.step.tasklet.single">
                <!-- (6) -->
                <batch:tasklet transaction-manager="jobTransactionManager"
                    ref="singleTransactionTasklet"/>
            </batch:step>
        </batch:flow>

        <batch:flow> <!-- (3) -->
            <batch:step id="parallelStepJob.step.chunk.file">
                <batch:tasklet transaction-manager="jobTransactionManager">
                    <!-- (7) -->
                    <batch:chunk reader="databaseReader" writer="fileWriter"
                        commit-interval="200"/>
                </batch:tasklet>
            </batch:step>
        </batch:flow>

    </batch:split>
</batch:job>
```

## 説明

項目番	説明
(1)	並列処理のために、各スレッドに割り当てるためのスレッドプールを定義する。
(2)	<batch:split>タグ内に並列実行するステップを<batch:flow>タグを使用した形式で定義をする。 task-executor属性に(1)で定義したスレッドプールのBeanを設定する
(3)	<batch:flow>ごとに並列位処理したい<batch:step>を定義する。
(4)	概要図のStep1：チャネルモデルの中間コミット方式処理を定義する。
(5)	概要図のStep2：タスクレットモデルの中間コミット方式処理を定義する。
(6)	概要図のStep3：タスクレットモデルの一括コミット方式処理を定義する。
(7)	概要図のStep4：チャネルモデルの非トランザクショナルなリソースに対する中間コミット方式処理を定義する。

### 並列処理したために処理性能が低下するケース

並列処理では多重処理同様にデータ範囲を変えて同じ処理を並列走行させることが可能である。この場合、データ範囲はパラメータなどで与える。

この際に、個々の処理ごとに対象となるデータ量が小さい場合、稼働時に占有するリソース量や処理時間などのフットプリントが並列処理では不利に働き、かえって処理性能が低下することがある。



### フットプリントの例

- 入力リソースに対するオープンから最初のデータ範囲を取得するまでの処理
  - リソースオープンは、データ取得に比べて処理時間がかかる
  - データ範囲のメモリ領域を初期化する処理も同様に時間がかかる

また、Parallel Stepの前後に共通処理のステップを定義することも可能である。

## 共通処理ステップを含むParallel Stepの例

```
<batch:job id="parallelRegisterJob" job-repository="jobRepository">
    <!-- (1) -->
    <batch:step id="parallelRegisterJob.step.preprocess"
        next="parallelRegisterJob.split">
        <batch:tasklet transaction-manager="jobTransactionManager"
            ref="deleteDetailTasklet" />
    </batch:step>

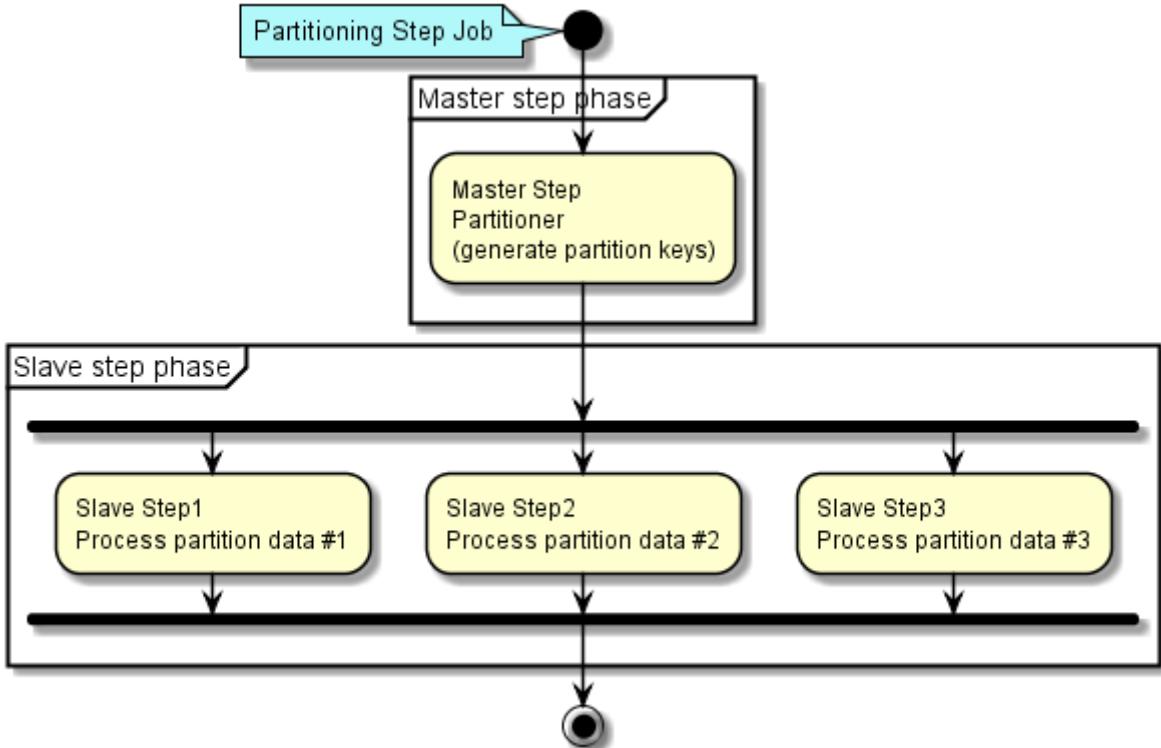
    <!--(2) -->
    <batch:split id="parallelRegisterJob.split" task-executor="parallelTaskExecutor">
        <batch:flow>
            <batch:step id="parallelRegisterJob.step.plan">
                <batch:tasklet transaction-manager="jobTransactionManager">
                    <batch:chunk reader="planReader" writer="planWriter"
                        commit-interval="1000" />
                </batch:tasklet>
            </batch:step>
        </batch:flow>
        <batch:flow>
            <batch:step id="parallelRegisterJob.step.performance">
                <batch:tasklet transaction-manager="jobTransactionManager">
                    <batch:chunk reader="performanceReader" writer="performanceWriter"
                        commit-interval="1000" />
                </batch:tasklet>
            </batch:step>
        </batch:flow>
    </batch:split>
</batch:job>
```

### 説明

項番	説明
(1)	前処理として処理するステップを定義する。next属性に<batch:split>に設定したidを指定する。 next属性による後続ステップの指定に関する詳細は <a href="#">シーケンシャルフロー</a> を参照。
(2)	Parallel Stepを定義する。 <batch:flow>ごとに並列処理したい<batch:step>を定義する。

### 8.2.2.2. Partitioning Step (多重処理)

Partitioning Step(多重処理)の方法を説明する



*Partitioning Step*の概要図

#### 概要図の説明

Partitioning Stepでは、MasterステップとSlaveステップの処理フェーズに分割される。

1. Masterステップでは、**Partitioner**により各Slaveステップが処理するデータ範囲を特定するための**Partition Key**を生成する。 **Partition Key**はステップコンテキストに格納される。
2. Slaveステップでは、ステップコンテキストから自身に割り当てられた**Partition Key**を取得し、それを使い処理対象データを特定する。特定した処理対象データに対して定義したステップの処理を実行する。

Partitioning Stepでは処理データを分割必要があるが、分割数については可変数と固定数のどちらにも対応できる。

#### 分割数

##### 可変数の場合

部門別で分割や、特定のディレクトリに存在するファイル単位での処理

##### 固定数の場合

全データを個定数で分割してデータを処理

Spring Batchでは、固定数のことを**grid-size**といい、**Partitioner**で**grid-size**になるようにデータ分割範囲を決定する。

Partitioning Stepでは、分割数をスレッドサイズより大きくすることができる。この場合、スレッド数分で多重実行され、スレッドに空きが出るまで、処理が未実行のまま保留となるステップが発生する。

以下にPartitioning Stepのユースケースを示す。

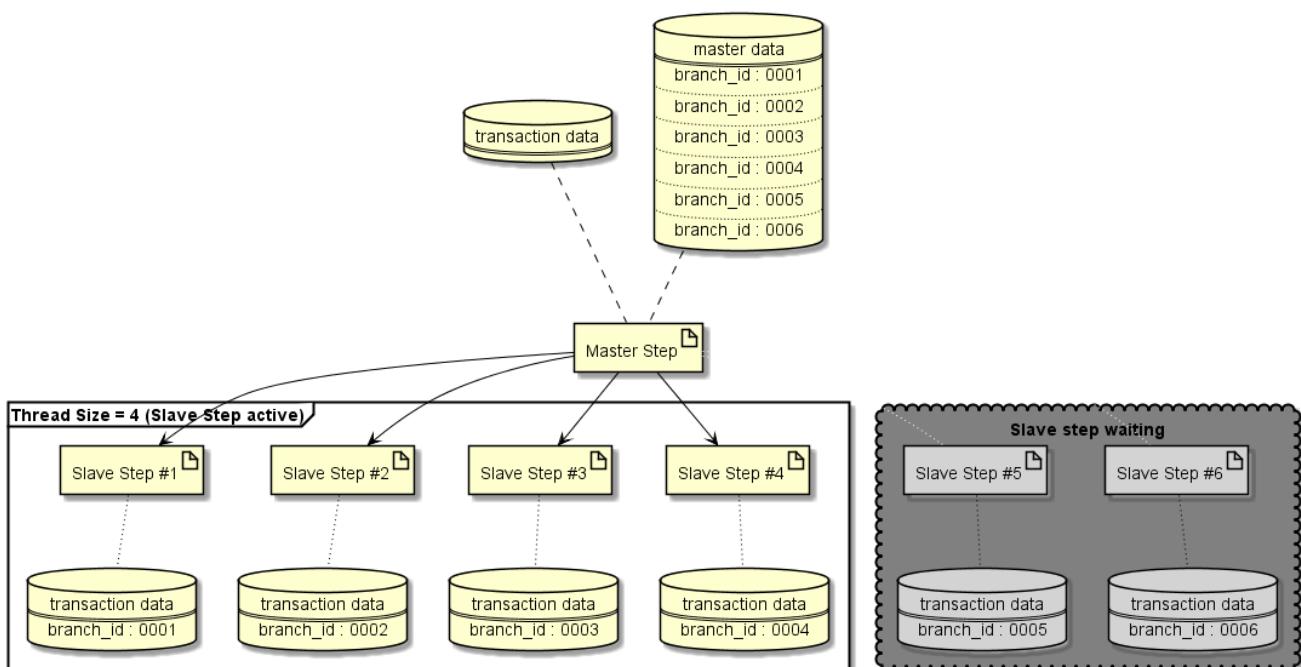
#### *Partitioning Step*のユースケース

ユースケース	Master(Partitioner)	Slave	分割数
マスタ情報からトランザクション情報を分割・多重化するケース 部門別や月別の集計処理など	DB(マスタ情報)	DB(トランザクション情報)	可変
複数ファイルのリストから1ファイル単位に多重化するケース 各支店からの転送データを支店別に多重処理(支店別集計処理など)	複数ファイル	單一ファイル	可変
大量データを一定数で分割・多重化するケース 障害発生時にリラン以外のリカバリ設計が難しくなるため、実運用では利用されることはないケース。 リランする場合は、全件やり直しなので分割したメリットが薄れてしまう。	grid-sizeとトランザクション情報件数からデータ範囲を特定	DB(トランザクション情報)	固定

#### 8.2.2.2.1. 分割数が可変の場合

Partitioning Stepで分割数を可変とする方法を説明する。

下記に処理イメージ図を示す。



処理イメージ図

処理イメージを例とした実装方法を示す。

## Repository(SQLMapper)の定義 (PostgreSQL)

```
<!-- (1) -->
<select id="findAll"
resultType="org.terasoluna.batch.functionalttest.app.model.mst.Branch">
  <![CDATA[
    SELECT
      branch_id AS branchId,
      branch_name AS branchName,
      branch_address AS branchAddress,
      branch_tel AS branchTel,
      create_date AS createDate,
      update_date AS updateDate
    FROM
      branch_mst
  ]]>
</select>

<!-- (2) -->
<select id="summarizeInvoice"

resultType="org.terasoluna.batch.functionalttest.app.model.performance.SalesPerformance
Detail">
  <![CDATA[
    SELECT
      branchId, year, month, customerId, SUM(amount) AS amount
    FROM (
      SELECT
        t2.charge_branch_id AS branchId,
        date_part('year', t1.invoice_date) AS year,
        date_part('month', t1.invoice_date) AS month,
        t1.customer_id AS customerId,
        t1.invoice_amount AS amount
      FROM invoice t1
      INNER JOIN customer_mst t2 ON t1.customer_id = t2.customer_id
      WHERE
        t2.charge_branch_id = #{branchId}
    ) t3
    GROUP BY branchId, year, month, customerId
    ORDER BY branchId ASC, year ASC, month ASC, customerId ASC
  ]]>
</select>

<!-- omitted -->
```

## Partitionerの実装例

```
@Component
public class BranchPartitioner implements Partitioner {

    @Inject
    BranchRepository branchRepository; // (3)

    @Override
    public Map<String, ExecutionContext> partition(int gridSize) {

        Map<String, ExecutionContext> map = new HashMap<>();
        List<Branch> branches = branchRepository.findAll();

        int index = 0;
        for (Branch branch : branches) {
            ExecutionContext context = new ExecutionContext();
            context.putString("branchId", branch.getBranchId()); // (4)
            map.put("partition" + index, context); // (5)
            index++;
        }

        return map;
    }
}
```

## Bean定義

```
<!-- (6) -->
<task:executor id="parallelTaskExecutor"
    pool-size="${thread.size}" queue-capacity="10"/>

<!-- (7) -->
<bean id="reader" class="org.mybatis.spring.batch.MyBatisCursorItemReader"
scope="step"

p:queryId="org.terasoluna.batch.functionalttest.app.repository.performance.InvoiceRepository.summarizeInvoice"
    p:sqlSessionFactory-ref="jobSqlSessionFactory">
<property name="parameterValues">
    <map>
        <!-- (8) -->
        <entry key="branchId" value="#{stepExecutionContext[branchId]}" />
    </map>
</property>
</bean>

<!-- omitted -->

<batch:job id="multipleInvoiceSummarizeJob" job-repository="jobRepository">
    <!-- (9) -->
    <batch:step id="multipleInvoiceSummarizeJob.master">
        <!-- (10) -->
        <batch:partition partitioner="branchPartitioner"
            step="multipleInvoiceSummarizeJob.slave">
            <!-- (11) -->
            <batch:handler grid-size="0" task-executor="parallelTaskExecutor" />
        </batch:partition>
    </batch:step>
</batch:job>

<!-- (12) -->
<batch:step id="multipleInvoiceSummarizeJob.slave">
    <batch:tasklet transaction-manager="jobTransactionManager">
        <batch:chunk reader="reader" writer="writer" commit-interval="10"/>
    </batch:tasklet>
</batch:step>
```

## 説明

項番	説明
(1)	マスタデータから処理対象を取得するSQLを定義する。
(2)	マスタデータからの取得値を検索条件とするSQLを定義する。
(3)	定義したRepository(SQLMapper)をInjectする。

項目番	説明
(4)	1つのSlaveステップが処理するマスタ値をステップコンテキストに格納する。
(5)	各Slaveが該当するコンテキストを取得できるようMapに格納する。
(6)	多重処理でSlaveステップの各スレッドに割り当てるためのスレッドプールを定義する。 Masterステップはメインスレッドで処理される。
(7)	マスタ値によるデータ取得のItemReaderを定義する。
(8)	(4)で設定したマスタ値をステップコンテキストから取得し、検索条件に追加する。
(9)	Masterステップを定義する。
(10)	データの分割条件を生成する処理を定義する。 <code>partitioner</code> 属性には、Partitionerインターフェース実装を設定する。 <code>step</code> 属性には、(12)で定義するSlaveステップのBeanIDを設定する。
(11)	<code>partitioner</code> では <code>grid-size</code> を使用しないため、 <code>grid-size</code> 属性には任意の値を設定する。 <code>task-executor</code> 属性に(6)で定義したスレッドプールのBeanIDを設定する。
(12)	Slaveステップを定義する。 <code>reader</code> 属性に(7)で定義したItemReaderを設定する。

複数ファイルのリストから1ファイル単位に多重化する場合は、Spring Batchが提供している以下のPartitionerを利用することができる。

- `org.springframework.batch.core.partition.support.MultiResourcePartitioner`

`MultiResourcePartitioner`の利用例を以下に示す。

## ファイル処理を多重化する例

```
<!-- (1) -->
<task:executor id="parallelTaskExecutor" pool-size="10" queue-capacity="200"/>

<!-- (2) -->
<bean id="reader"
    class="org.springframework.batch.item.file.FlatFileItemReader" scope="step"
    p:resource="#{stepExecutionContext.fileName}"> <!-- (3) -->
    <property name="lineMapper">
        <bean class="org.springframework.batch.item.mapping.DefaultLineMapper"
            p:fieldSetMapper-ref="invoiceFieldSetMapper">
            <property name="lineTokenizer">
                <bean
                    class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer"
                    p:names="invoiceNo,salesDate,productId,customerId,quant,price"/>
            </property>
        </bean>
    </property>
</bean>

<!-- (4) -->
<bean id="partitioner"

    class="org.springframework.batch.core.partition.support.MultiResourcePartitioner"
    scope="step"
    p:resources="file:#{jobParameters[basedir]}/input/invoice-* .csv"/> <!-- (5) -->

<!--(6) -->
<batch:job id="inspectPartitioninglStepFileJob" job-repository="jobRepository">
    <batch:step id="inspectPartitioninglStepFileJob.step.master">
        <batch:partition partitioner="partitioner"
            step="inspectPartitioninglStepFileJob.step.slave">
            <batch:handler grid-size="0" task-executor="parallelTaskExecutor"/>
        </batch:partition>
    </batch:step>
</batch:job>

<!-- (7) -->
<batch:step id="inspectPartitioninglStepFileJob.step.slave">
    <batch:tasklet>
        <batch:chunk reader="reader" writer="writer" commit-interval="20"/>
    </batch:tasklet>
</batch:step>
```

### 説明

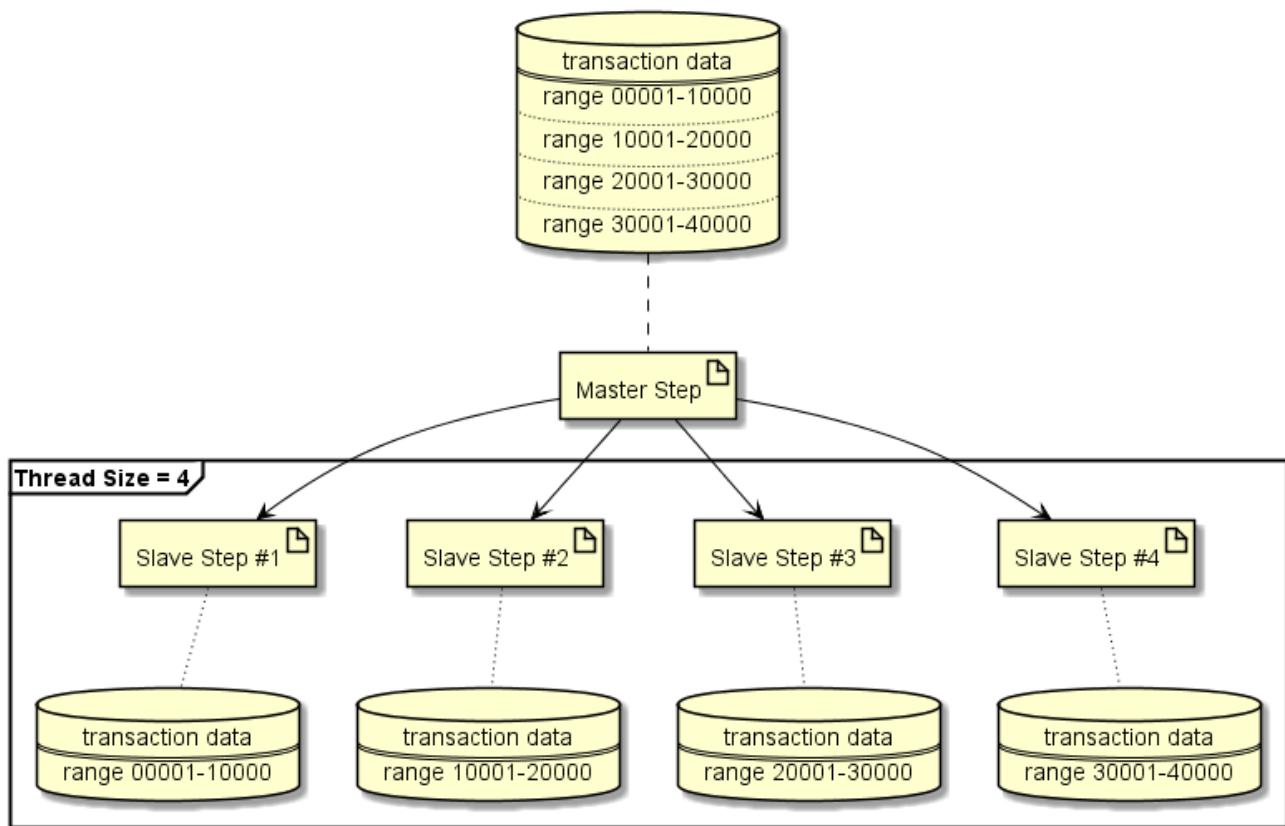
項番	説明
(1)	多重処理でSlaveステップの各スレッドに割り当てるためのスレッドプールを定義する。 Masterステップはメインスレッドで処理される。

項目番	説明
(2)	1つのファイルを読み込むためのItemReaderを定義する。
(3)	resourceプロパティに、MultiResourcePartitionerで分割されたファイルを入力ファイルに指定する。 MultiResourcePartitionerは、"fileName"というキーでステップコンテキストにファイルパスを格納している。
(4)	MultiResourcePartitionerをPartitionerとして定義する。
(5)	*を用いたパターンを使用することで、複数ファイルを対象にすることができる。
(6)	Masterステップを定義する。 定義内容は上記で説明したPartitioning Stepの内容と同じ。
(7)	Slaveステップを定義する。 reader属性に(2)で定義したItemReaderを設定する。

#### 8.2.2.2.2. 分割数が固定の場合

Partitioning Stepで分割数を固定する方法を説明する。

下記に処理イメージ図を示す。



処理イメージ図

処理イメージを例とした実装方法を示す。

## Repository(SQLMapper)の定義 (PostgreSQL)

```
<!-- (1) -->
<select id="findByYearAndMonth"

resultType="org.terasoluna.batch.functionalttest.app.model.performance.SalesPerformance
Summary">
  <![CDATA[
    SELECT
      branch_id AS branchId, year, month, amount
    FROM
      sales_performance_summary
    WHERE
      year = #{year} AND month = #{month}
    ORDER BY
      branch_id ASC
    LIMIT
      #{dataSize}
    OFFSET
      #{offset}
  ]]>
</select>

<!-- (2) -->
<select id="countByYearAndMonth" resultType="_int">
  <![CDATA[
    SELECT
      count(*)
    FROM
      sales_performance_summary
    WHERE
      year = #{year} AND month = #{month}
  ]]>
</select>

<!-- omitted -->
```

## Partitionerの実装例

```
@Component
public class SalesDataPartitioner implements Partitioner {

    @Inject
    SalesSummaryRepository repository; // (3)

    // omitted

    @Override
    public Map<String, ExecutionContext> partition(int gridSize) {

        Map<String, ExecutionContext> map = new HashMap<>();
        int count = repository.countByYearAndMonth(year, month);
        int dataSize = (count / gridSize) + 1;           // (4)
        int offset = 0;

        for (int i = 0; i < gridSize; i++) {
            ExecutionContext context = new ExecutionContext();
            context.putInt("dataSize", dataSize);          // (5)
            context.putInt("offset", offset);             // (6)
            offset += dataSize;
            map.put("partition:" + i, context);           // (7)
        }

        return map;
    }
}
```

## Bean定義

```
<!-- (8) -->
<task:executor id="parallelTaskExecutor"
    pool-size="${thread.size}" queue-capacity="10"/>

<!-- (9) -->
<bean id="reader"
    class="org.mybatis.spring.batch.MyBatisCursorItemReader" scope="step"

    p:queryId="org.terasoluna.batch.functionalttest.ch08.parallelandmultiple.repository.Sal
    esSummaryRepository.findByYearAndMonth"
    p:sqlSessionFactory-ref="jobSqlSessionFactory">
    <property name="parameterValues">
        <map>
            <entry key="year" value="#{new Integer(jobParameters[year])}" />
            <entry key="month" value="#{new Integer(jobParameters[month])}" />
            <!-- (10) -->
            <entry key="dataSize" value="#{stepExecutionContext[dataSize]}" />
            <!-- (11) -->
            <entry key="offset" value="#{stepExecutionContext[offset]}" />
        </map>
    </property>
</bean>

<!-- omitted -->

<batch:job id="multipleCreateSalesPlanSummaryJob" job-repository="jobRepository">
    <!-- (12) -->
    <batch:step id="multipleCreateSalesPlanSummaryJob.master">
        <!-- (13) -->
        <batch:partition partitioner="salesDataPartitioner"
            step="multipleCreateSalesPlanSummaryJob.slave">
            <!-- (14) -->
            <batch:handler grid-size="4" task-executor="parallelTaskExecutor" />
        </batch:partition>
    </batch:step>
</batch:job>

<!-- (15) -->
<batch:step id="multipleCreateSalesPlanSummaryJob.slave">
    <batch:tasklet transaction-manager="jobTransactionManager">
        <batch:chunk reader="reader" processor="addProfitsItemProcessor"
            writer="writer" commit-interval="10"/>
    </batch:tasklet>
</batch:step>
```

## 説明

項目番	説明
(1)	特定のデータ範囲を取得するためにページネーション検索(SQL絞り込み方式)を定義する。 ページネーション検索(SQL絞り込み方式)の詳細は、TERASOLUNA Server 5.x 開発ガイドラインの <a href="#">Entityのページネーション検索(SQL絞り込み方式)</a> を参照。
(2)	処理対象の全件数を取得するSQLを定義する。
(3)	定義したRepository(SQLMapper)をInjectする。
(4)	1つのSlaveステップが処理するデータ件数を算出する。
(5)	(4)のデータ件数をステップコンテキストに格納する。
(6)	各Slaveステップの検索開始位置をステップコンテキストに格納する。
(7)	各Slaveが該当するコンテキストを取得できるようMapに格納する。
(8)	多重処理でSlaveステップの各スレッドに割り当てるためのスレッドプールを定義する。 Masterステップはメインスレッドで処理される。
(9)	ページネーション検索(SQL絞り込み方式)によるデータ取得のItemReaderを定義する。
(10)	(5)で設定したデータ件数をステップコンテキストから取得し、検索条件に追加する。
(11)	(6)で設定した検索開始位置をステップコンテキストから取得し、検索条件に追加する。
(12)	Masterステップを定義する。
(13)	データの分割条件を生成する処理を定義する。 <code>partitioner</code> 属性には、 <a href="#">Partitioner</a> インターフェース実装を設定する。 <code>step</code> 属性には、(15)で定義するSlaveステップのBeanIDを設定する。
(14)	<code>grid-size</code> 属性に分割数(固定値)を設定する。 <code>task-executor</code> 属性に(8)で定義したスレッドプールのBeanIDを設定する。
(15)	Slaveステップを定義する。 <code>reader</code> 属性に(9)で定義したItemReaderを設定する。

# Chapter 9. 利用時の注意点

## 9.1. TERASOLUNA Batch 5.xの注意点について

ここでは、各節で説明しているTERASOLUNA Batch 5.xを利用する際の、ルールや注意点についてリストにまとめる。ユーザはバッチアプリケーションを開発する際、以降に示すポイントに留意して進めてほしい。



ここでは、特に重要な注意点を挙げているのみであり、あらゆる検討事項を網羅しているわけではない。ユーザは必ず利用する機能を一読すること。

### バッチ処理で考慮する原則と注意点

- 単一のバッチ処理は可能な限り簡素化し、複雑な論理構造を避ける。
- 複数のジョブで同じことを何度もしない。
- システムリソースの利用を最小限にし、不要な物理I/Oを避け、メモリ上での操作を活用する。

### TERASOLUNA Batch 5.xの指針

- [バッチアプリケーションの開発](#)
  - 1ジョブ=1Bean定義(1ジョブ定義)として作成する
  - 1ステップ=1バッチ処理=1ビジネスロジックとして作成する
- [チャックモデル](#)
  - 大量データを効率よく処理したい場合に利用する。
- [タスクレットモデル](#)
  - シンプルな処理や、定型化しにくい処理、データを一括で処理したい場合に利用する。
- [同期実行](#)
  - スケジュールどおりにジョブを起動したり、複数のジョブを組み合わせてバッチ処理を行う場合に利用する。
- [非同期実行\(DBポーリング\)](#)
  - ディレード処理、処理時間が短いジョブの連続実行、大量ジョブの集約などに利用する。
- [非同期実行\(Webコンテナ\)](#)
  - DBポーリングと同様だが、起動までの即時性が求められる場合にはこちらを利用する。
- JobRepositoryの管理
  - Spring Batch はジョブの起動状態・実行結果の記録にJobRepositoryを使用する。
  - TERASOLUNA Batch 5.xでは、以下のすべてに該当する場合は永続化は任意としてよい。

- 同期型ジョブ実行のみでTERASOLUNA Batch 5.xを使用する。
- ジョブの停止・リスタートを含め、ジョブの実行管理はすべてジョブスケジューラに委ねる。
  - Spring BatchがもつJobRepositoryを前提としたリスタートを利用しない。
- これらに該当する場合はJobRepositoryが使用するRDBMSの選択肢として、インメモリ・組み込み型データベースであるH2を利用する。一方で非同期実行を利用する場合や、Spring Batchの停止・リスタートを活用する場合は、ジョブの実行状態・結果を永続化可能なRDBMSが必要となる。  
この点については、[ジョブの管理](#)も一読のこと。

---

## チャンクモデルとタスクレットモデルの使い分け

- [チャンクモデル](#)
  - 大量のデータを安定して処理したい場合
  - 件数ベースリスタートをしたい場合
- [タスクレットモデル](#)
  - リカバリを限りなくシンプルにしたい場合
  - 処理の内容をまとめたい場合

[チャンクモデルとタスクレットモデルの使い分け](#)も一読のこと。

---

## Beanスコープの統一

- Tasklet実装では、Injectされるコンポーネントのスコープに合わせる。
- Composite系コンポーネントは、委譲するコンポーネントのスコープに合わせる。
- JobParameterを使用する場合は、stepのスコープにする。
- Step単位でインスタンス変数を確保したい場合は、stepのスコープにする。

---

## 性能チューニングポイント

- チャンクサイズを調整する
  - チャンクを利用するときは、コミット件数を適度なサイズにする。サイズを大きくしすぎない。
- フェッチサイズを調整する
  - データベースアクセスでは、フェッチサイズを適度なサイズにする。サイズを大きくしすぎない。
- ファイル読み込みを効率化する
  - 専用のFieldSetMapperインターフェース実装を用意する。
- 並列処理・多重処理

- 出来る限りジョブスケジューラによって実現する。
  - 分散処理
    - 出来る限りジョブスケジューラによって実現する。
- 

### 非同期実行(DBポーリング)

- インメモリデータベースの使用
    - 長期連続運用するには向かず、定期的に再起動する運用が望ましい。
    - 長期連続運用で利用したい場合は、定期的にJobRepositoryからデータを削除するなどのメンテナンス作業が必須である。
  - 登録ジョブの絞込み
    - 非同期実行することを前提に設計・実装されたジョブを指定する。
  - 性能劣化もあり得るため、超ショートバッチの大量処理は向いていない。
  - 同一ジョブの並列実行が可能になっているので、並列実行した場合に同一ジョブが影響を与えないようにする必要がある
- 

### 非同時実行(Webコンテナ)

- 基本的な検討事項は、[非同期実行\(DBポーリング\)](#)と同じ。
  - スレッドプールの調整をする。
    - 非同期実行のスレッドプールとは別に、Webコンテナのリクエストスレッドや同一筐体内で動作している他のアプリケーションも含めて検討する必要がある。
  - Webとバッチでは、データソース、MyBatis設定、Mapperインターフェースは相互参照はできない。
  - スレッドプール枯渀によるジョブの起動失敗はジョブ起動時に補足できないので、別途確認する手段を用意しておく。
- 

### データベースアクセスとトランザクション

- 「ItemWriterにMyBatisBatchItemWriterを使用する」と「ItemProcessorでMapperインターフェースを使用し参照更新をする」は同時にできない。
  - MyBatisには、同一トランザクション内で2つ以上のExecutorTypeで実行してはいけないという制約があるため。 [ItemReader・ItemWriter以外のデータベースアクセス](#)を参照。
- データベースの同一テーブルへ入出力する際の注意点
  - 読み取り一貫性を担保するための情報が出力(UPDATEの発行)により失われた結果、入力(SELECT)にてエラーが発生することがある。以下の対策を検討する。
    - データベースに依存になるが、情報を確保する領域を大きくする。

入力データを分割し多重処理を行う。

---

## ファイルアクセス

- 以下の固定長ファイルを扱う場合は、TERASOLUNA Batch 5.xが提供する部品を必ず使う。
    - マルチバイト文字を含む固定長ファイル
    - 改行なし固定長ファイル
  - フッタレコードを読み飛ばす場合は、OSコマンドによる対応が必要。
- 

## 排他制御

- 複数ジョブを同時実行する場合は、排他制御の必要がないようにジョブ設計を行う。
    - アクセスするリソースや処理対象をジョブごとに分割することが基本である。
  - デッドロックが発生しないように設計を行う。
  - ファイルの排他制御は、タスクレットモデルで実装すること。
- 

## 異常系への対応

- 例外ハンドリングではトランザクション処理を行わない。
  - 処理モデルでChunkListenerの挙動が異なることに注意する。
    - リソースのオープン・クローズで発生した例外は、
      - チャンクモデル...ChunkListenerインターフェースが捕捉するスコープ外となる。
      - タスクレットモデル...ChunkListenerインターフェースが捕捉するスコープ内となる。
  - 入力チェックエラーは、チェックエラーの原因となる入力リソースを修正しない限り、リスタートしても回復不可能である
  - JobRepositoryに障害が発生した時の対処方法を検討する必要がある。
- 

## *ExecutionContext*について

- ExecutionContext*はJobRepositoryへ格納されるため、以下の制約がある。
    - ExecutionContext*へ格納するオブジェクトは、`java.io.Serializable`を実装したクラスでなければならぬ。
    - 格納できるサイズに制限がある。
- 

## 終了コード

- Javaプロセスの強制終了とバッチアプリケーションの終了状態とは明確に区別する。
-

- バッチアプリケーションによるプロセスの終了コードを1に設定することは厳禁とする。
- 

## 並列処理と多重処理

- **Multi Thread Step**は利用しない。
- 処理内容によっては、リソース競合とデッドロックが発生する可能性に注意する。