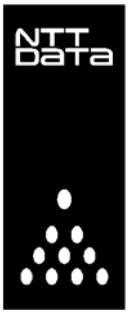


TERASOLUNA Batch Framework for Java

機能説明書

第 2.0.1.0 版

株式会社 NTT データ

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、「TERASOLUNA Batch Framework for Java (機能説明書)」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。

Java, JDK, J2SE, J2EE は、米国 Sun Microsystems, Inc.の米国及びその他の国における登録商標または商標です。

Oracle は、米国 Oracle International Corp.の米国及びその他の国における登録商標または商標です。

TERASOLUNA は、株式会社 NTT データの登録商標です。

WebLogic は、BEA Systems Inc.の登録商標または商標です。

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。

本書は、TERASOLUNA Batch Framework for Java ver2.0.1.0 に対応しています。

目次

- TERASOLUNA Batch Framework for Java　機能一覧

BA-01 トランザクション管理機能

■ 概要

◆ 機能概要

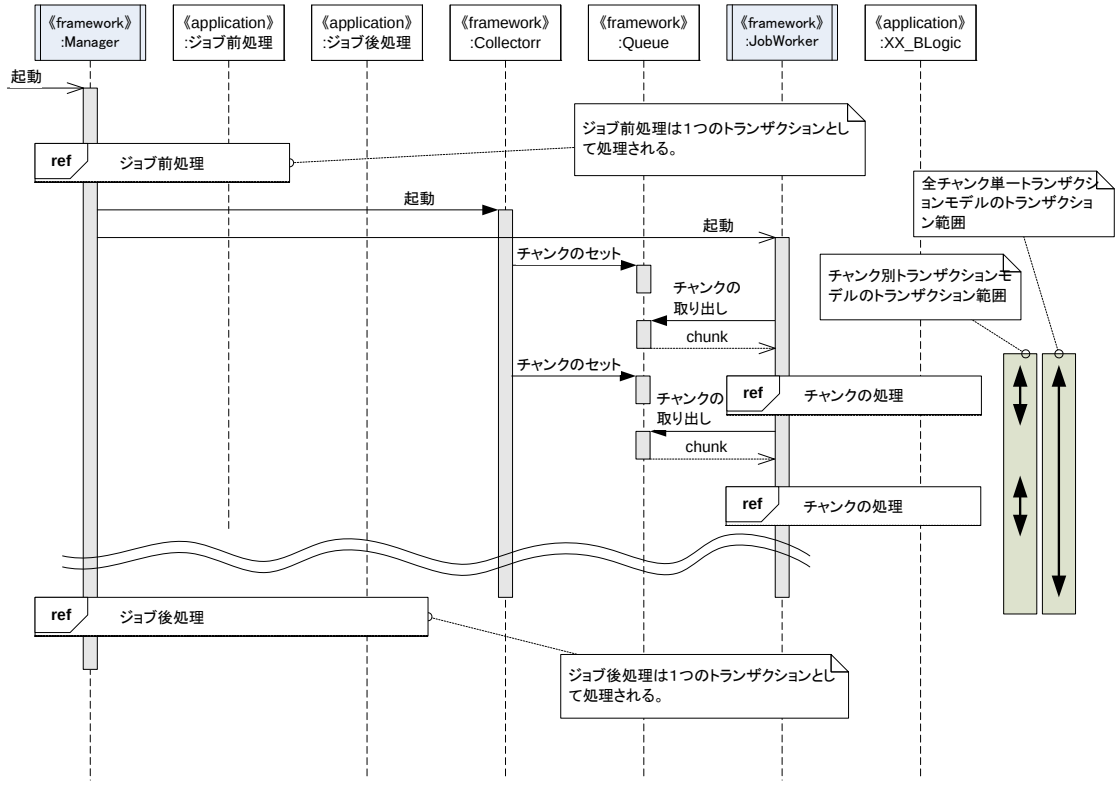
- トランザクション管理機能を提供する。
  - フレームワークでトランザクション制御を行なうため、開発者がトランザクションコードを実装する必要がない。
  - コミット・ロールバックはフレームワークが行なう。

- 対象データを単位に繰り返し起動されるビジネスロジックに対して、指定件数毎にまとめてコミットを発行する機能を提供する。
  - 開発者は、環境、あるいはジョブの特性に応じて、何件毎にコミットを行うかを設定ファイルで指定する。

- 以下の３つのトランザクションモデルをフレームワークで提供する。開発者は、ジョブに適したトランザクションモデルを選択する。
  1. チャンク別トランザクションモデル
     - チャンク単位に、トランザクションで処理する。チャンクは、指定された件数のデータを含むか、あるいはコントロールブレイクを単位に作成される。
  2. 全チャンク単一トランザクションモデル
     - すべてのチャンクを単一のトランザクションで処理する。
  3. 非トランザクションモデル
     - フレームワークがトランザクション管理を行わず開発者にゆだねるモデルである。ファイルのみを扱うジョブなどでトランザクション管理の必要のないジョブで選択する。

◆ 概念図

- トランザクションモデルのトランザクション範囲
  フレームワークで用意するそれぞれのトランザクションモデル毎に、以下のトランザクション範囲が設定されている。



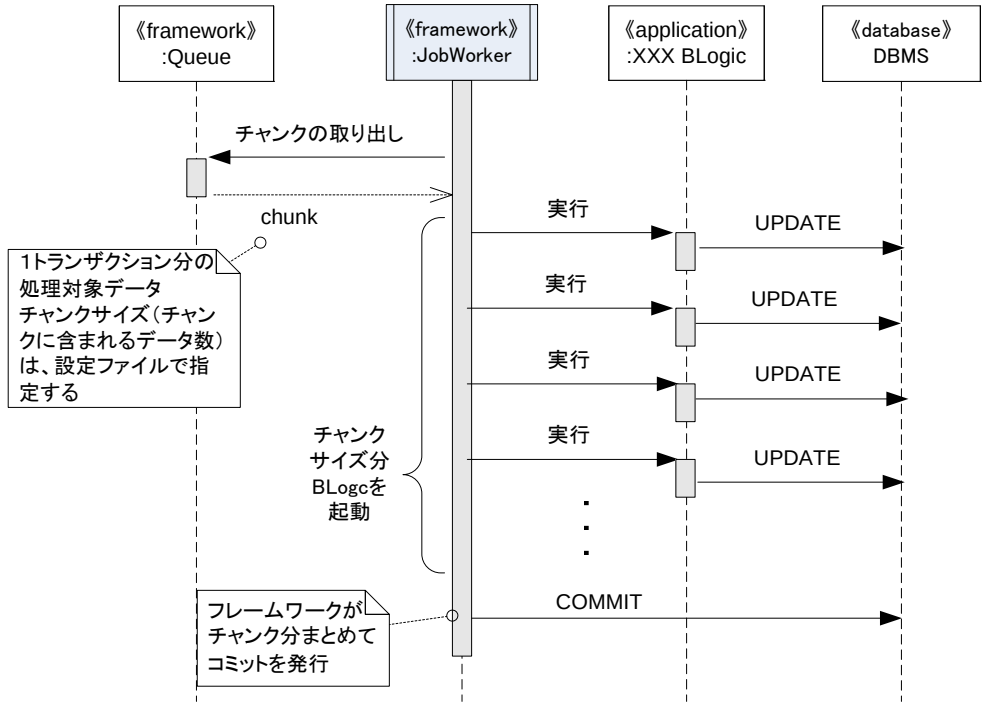
項番	トランザクション モデル	トランザクション範囲
1	チャンク別トランザクションモデル	各チャンクは、それぞれ別のトランザクションとして処理される。
2	全チャンク単一トランザクションモデル	全てのチャンクは、単一のトランザクションで処理される。
3	非トランザクションモデル	なし

◆ 解説

- ジョブの種類とトランザクションモデル
  トランザクションモデルは、通常ジョブ（分割されていないジョブ）、子ジョブ（分割ジョブで分割されたそれぞれの部分を実行するジョブ）に対して選択する。トランザクションモデルは、データ件数、エラー時のリカバリ方法などを考慮して選択する。
  親ジョブ（分割ジョブで分割を行うジョブ）には対象データの処理がないため、トランザクションモデルを選択することができない。全ての子ジョブを単一のトランザクションとすることはできない。

項番	ジョブの種類		選択可能なトランザクションモデル	選択基準
1	通常ジョブ		チャンク別トランザクションモデル	通常は、このトランザクションモデルを選択する
2			全チャンク単一トランザクションモデル	処理対象のデータ件数が少なく、1トランザクションで処理しても DB などのリソースに問題ないとき
3			非トランザクションモデル	ファイルアクセスのみなど、トランザクション管理が必要ないとき
4	分割ジョブ	親ジョブ	選択不可	—
5		子ジョブ	チャンク別トランザクションモデル	通常ジョブと同じ
6			全チャンク単一トランザクションモデル	通常ジョブと同じ
7			非トランザクションモデル	通常ジョブと同じ

- フレームワークによる指定件数毎のコミット
  バッチ処理では、対象データの一件毎にコミットを発行すると性能が得られないため、100 件、200 件等の複数件に対してまとめてコミットを発行する。



フレームワークによるコミットは、チャンクを単位に行う。チャンクに含まれるデータ全体が、DBMS へコミットする単位となる。フレームワークが何件毎にまとめてコミットを発行するかは、フレームワークの「チャンクサイズ」パラメータで設定する。

- ジョブ前処理とジョブ後処理のトランザクション
  ジョブ前処理とジョブ後処理は、チャンクを処理するトランザクションとは独立して処理される。
  - 複数のジョブ前処理、ジョブ後処理を指定した場合には、単一のトランザクションで処理される。
  - 分割ジョブである場合には、親ジョブと子ジョブのどちらに対しても、ジョブ前処理とジョブ後処理が指定できる。
  - 非トランザクションモデルのジョブ前処理、ジョブ後処理においてデータベースアクセスを行う際には、アプリケーション開発者がトランザクション設定を行うこと。
  - ジョブ前処理、ジョブ後処理についての詳細は、『BD-01 ビジネスロジック実行機能』を参照のこと。

- 全チャンク単一トランザクションモデルでの先頭チャンク前処理と最終チャンク後処理のトランザクション
  - 先頭チャンク前処理、最終チャンク後処理のどちらも、全てのチャンクを処理するトランザクションの中で実行される。
  - 先頭チャンク前処理、最終チャンク後処理についての詳細は、『BD-01 ビジネスロジック実行機能』を参照のこと。

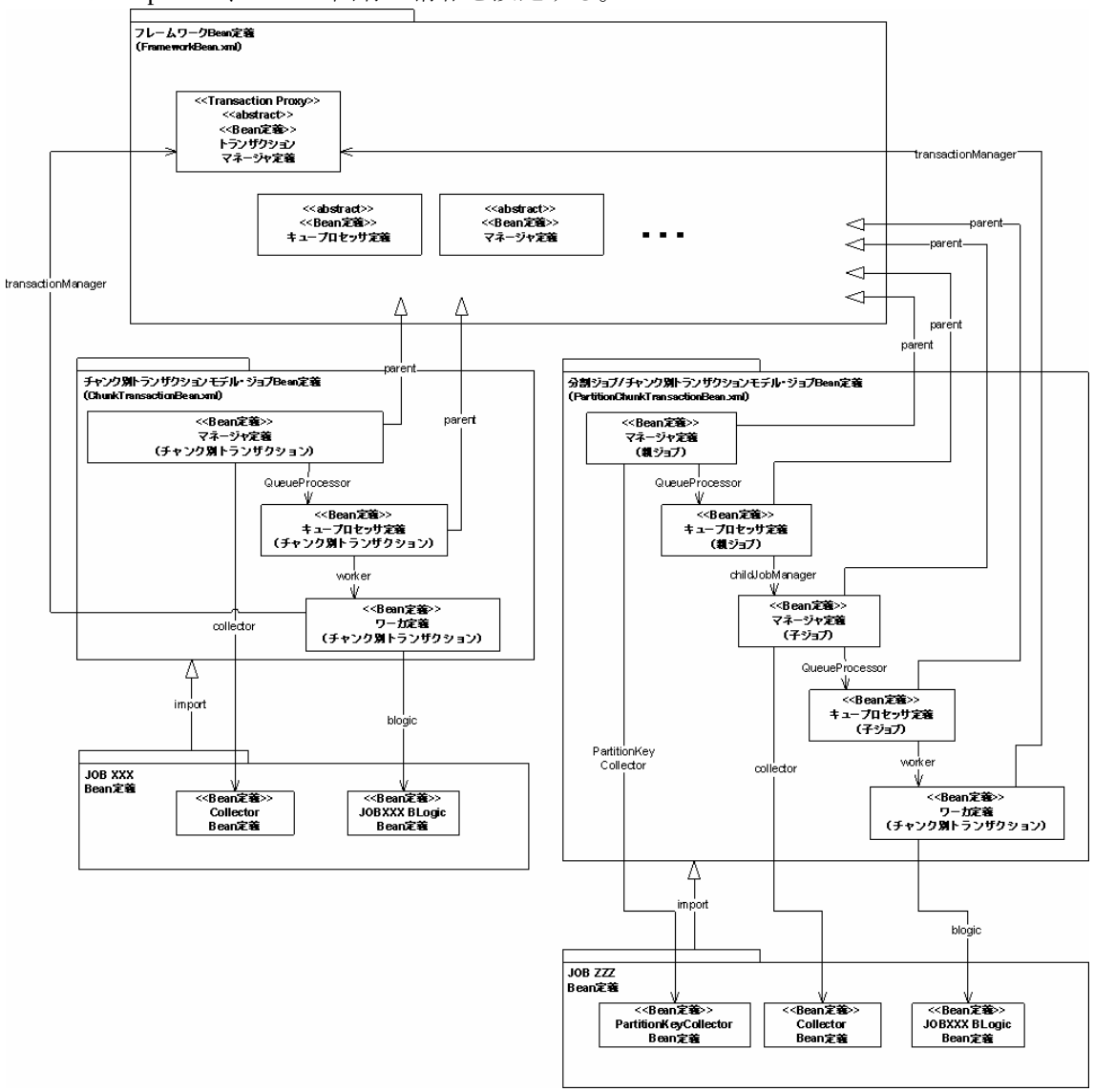
- ジョブ前処理／ジョブ後処理と、先頭チャンク前処理／最終チャンク後処理のトランザクションモデルとの関係
  ジョブ前処理／ジョブ後処理は、ジョブの種類、トランザクションモデルに関わらず、すべてのジョブに対して設定できる。
  先頭チャンク前処理／最終チャンク後処理は、全チャンク単一トランザクションモデルである場合のみ設定できる。トランザクションモデルが指定できない親ジョブ、およびチャンク別トランザクションモデルのジョブに対しては設定できない。

項番	ジョブの種類		選択可能なトランザクションモデル	設定可能な前／後処理	
				ジョブ前処理、ジョブ後処理	先頭チャンク前処理、最終チャンク後処理
1	通常ジョブ		チャンク別トランザクションモデル	○	×
2			全チャンク単一トランザクションモデル	○	○
3			非トランザクションモデル	○ (データベースアクセスがない処理)	×
4	分割ジョブ	親ジョブ	選択不可	○	×
5		子ジョブ	チャンク別トランザクションモデル	○	×
6			全チャンク単一トランザクションモデル	○	○
7			非トランザクションモデル	○ (データベースアクセスがない処理)	×

- ジョブ前処理／ジョブ後処理と、先頭チャンク前処理／最終チャンク後処理のトランザクションの違い
  ジョブ前処理／ジョブ後処理をジョブに対して複数の処理を指定した場合には、ジョブ前処理／ジョブ後処理はそれぞれ単一のトランザクションで処理される。
  先頭チャンク前処理／最終チャンク後処理は、対象データに対する処理と同一のトランザクションで処理される。

項番	処理の種類	データ処理のトランザクションとの関係	複数処理を指定した場合のトランザクション
1	ジョブ前処理／ジョブ後処理	チャンクを処理するトランザクションとは別のトランザクション	ジョブ前処理、ジョブ後処理でそれぞれ一つのトランザクション
2	先頭チャンク前処理／最終チャンク後処理	チャンクを処理するトランザクションと同一のトランザクション	複数の処理を指定しても、チャンクを処理するトランザクションと同一のトランザクション

- フレームワーク Bean 定義ファイル、ジョブ Bean 定義ファイルの構造
  ジョブでどのビジネスロジックを使うのか等のジョブを定義するための情報はジョブ Bean に設定する。ジョブ Bean のための Java クラスは、フレームワークで用意されている。
  フレームワーク Bean 定義ファイルでは、ジョブに依存しない部分があらかじめ定義されている。個別のジョブを定義する際には、トランザクションモデルに対応したトランザクションモデル別 Bean 定義ファイル雛形をジョブ Bean 定義ファイルに import し、ジョブ固有の情報を設定する。



- 共通処理のトランザクション
  共通処理（ビジネスロジックから呼び出す処理）を呼び出し元とは別のトランザクションで処理したい場合には、共通処理に対するトランザクション設定をSpringAOP で行う。
  SpringAOP で設定できるトランザクションの詳細は、コーディングポイントを参照のこと。

- ファイル等の非トランザクションリソースの扱い
  フレームワークでは、ファイル出力などのトランザクショナルでないリソースに対しては、トランザクション処理は行わない。
  - ファイル出力でトランザクション的な処理を行いたい場合には、一時ファイルに書き出してから適切なタイミングで実際のファイルに移動する、あるいは DB などのトランザクショナルなリソースに出力し、処理の最後でファイルに落とすなどの処理を業務アプリケーションで行う。

- 分散トランザクション
  フレームワークでは分散トランザクションには対応しない。

- エラー処理の実装方針（エラー時のロールバックの扱い）
  フレームワークでは原則として複数件の処理対象データに対してまとめてコミットを発行する。そのため、エラー等でロールバックするとそれまで正常に処理したデータであっても、同じトランザクションで処理されているものも一緒にロールバックされる。
  したがって、業務アプリケーションではデータベースへの更新処理を行う前にエラーデータをチェックし、業務アプリケーションのロジックではロールバックの必要がないようにすること。

- セーブポイントの利用
  チャンク別トランザクションモデルでは、セーブポイントを利用することができる。セーブポイントを利用するためには、ジョブ Bean 定義ファイルでセーブポイント利用フラグを設定する。
  セーブポイントを利用するジョブでは、以下のようにエラー時のロールバック範囲が最小となるようにトランザクション処理が行われる。

項番	ビジネスロジックのリターンコード	トランザクション処理	
		セーブポイントを利用しない場合	セーブポイントを利用する場合
1	NORMAL_CONTINUE	ビジネスロジックの入力データがチャンクの最後のデータである場合には、コミットする。 チャンクの最後のデータではない場合には、何もしない。	ビジネスロジックの入力データがチャンクの最後のデータである場合には、コミットする。 チャンクの最後のデータではない場合には、保持していたセーブポイントをリリースし、新たにセーブポイントを作成する。
2	NORMAL_END	トランザクションをコミットする。	トランザクションをコミットする。
3	ERROR_CONTINUE	ビジネスロジックの入力データがチャンクの最後のデータである場合には、コミットする。 チャンクの最後のデータではない場合には、何もしない。 （NORMAL_CONTINUE の場合と同じ）	ビジネスロジックの入力データがチャンクの最後のデータである場合には、保持していたセーブポイントまでロールバックした後、コミットする。 チャンクの最後のデータではない場合には、保持していたセーブポイントまでロールバックする。
4	ERROR_END	トランザクションをロールバックする。	保持していたセーブポイントまでロールバックし、トランザクションをコミットする。

セーブポイントを利用する場合には、ERROR_CONTINUE がリターンされたビジネスロジック入力データに対する処理がトランザクション範囲に入らないようにセーブポイントが使用される。NORMAL_CONTINUE がリターンされるたびにセーブポイントが更新され、ERROR_CONTINUE がリターンされた場合には、前回 NORMAL_CONTINUE がリターンされたポイントまでトランザクションをロールバックする。
セーブポイントが利用できるのは、チャンク別トランザクションモデルだけである。全チャンク単一トランザクションモデルでは、ジョブ全体を一つのトランザクションとして処理するモデルであり、セーブポイントを使って一部のデータだけを更新する処理はそぐわないためサポートされない。
また、セーブポイントの利用はメインの処理となるビジネスロジックの実行のみ可能である。

■ 使用方法

◆ コーディングポイント

- トランザクションの種類（『CA-01 トランザクション管理機能』の再掲）
  トランザクションには、以下のような種類が存在するが、TERASOLUNA-Batch では基本的には“PROPAGATION_REQUIRED”を使用する。必要があれば各自検討し変更すること。

トランザクション	概要
PROPAGATION_REQUIRED	**現在のトランザクションをサポートし、トランザクションが存在しなければ新しいトランザクションを作成する。TERASOLUNA-Batch で標準的に使用する。**
PROPAGATION_SUPPORTS	現在のトランザクションをサポートし、トランザクションが存在しなければ非トランザクション的に実行する。
PROPAGATION_MANDATORY	現在のトランザクションをサポートし、トランザクションが存在しなければ例外を送出する。
PROPAGATION_REQUIRES_NEW	新しいトランザクションを作成し、現在のトランザクションが存在すればそれを一時停止する。
PROPAGATION_NOT_SUPPORTED	非トランザクション的に実行し、現在のトランザクションが存在すればそれを一時停止する。
PROPAGATION_NEVER	非トランザクション的に実行し、トランザクションが存在すれば例外を送出する。
PROPAGATION_NESTED	現在のトランザクションが存在すればネストされたトランザクションの内部で実行し、存在しなければ PROPAGATION_REQUIRED と同様に動作する。

Spring では、その他にも独立性レベルやタイムアウト、読取専用などの定義情報の設定が可能である。詳細は、SpringAPI を参照のこと。

- トランザクションマネージャとトランザクションプロキシファクトリ
  トランザクションマネージャには、Spring が提供する DatasourceTransactionManager を使用する。DatasourceTransactionManager は、単一の JDBC データソースに対してトランザクションを実行するトランザクションマネージャである。トランザクションモデル別 Bean 定義ファイルでは、トランザクションが設定された Worker があらかじめ設定されている。
  また、フレームワーク Bean 定義ファイルには、トランザクションマネージャを指定したファクトリクラス TransactionProxyFactoryBean が設定されている。
  DatasourceTransactionManager、TransactionProxyFactoryBean の詳細については、SpringAPI を参照のこと。

  - データアクセス Bean 定義ファイル（トランザクションマネージャ、トランザクションプロキシファクトリ設定関連部分）

```
<!-- DataSourceの設定。 -->
<bean id="dataSource" class="……">……</bean>

<!-- 単一のJDBCデータソース向けのトランザクションマネージャ。 -->
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>

<!-- トランザクションプロキシのベース定義となる親Bean定義。
  - 子のBean定義で属性をオーバーライドすることができる。-->
<bean id="baseTransactionProxy"
    class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
    abstract="true">
  <property name="transactionManager" ref="transactionManager" />
</bean>
```

transactionManager プロパティに DataSourceTransactionManager を指定する。

Spring で提供されているファクトリクラス。任意のクラス（target 属性で指定）に、トランザクション制御コードを付加したプロキシクラスを生成する。

プロキシ対象の Bean、およびトランザクション定義情報の設定は、この baseTransactionProxy を親定義とするフレームワークの Bean 定義で行われている。

- トランザクションモデル別 Bean 定義ファイル雛形の選択
  TERASOLUNA-Batch では、開発者がジョブ Bean を定義する際の雛形として利用できるトランザクションモデル別 Bean 定義が用意されており、あらかじめ必要なトランザクションが設定されている。開発者は、ジョブのトランザクションモデルに応じて雛形を選択して、ジョブ Bean 定義ファイルを作成する。
  フレームワークで用意している雛型には、以下のものがある。

項番	トランザクションモデル別 Bean 定義ファイル名	概要
1	ChunkTransactionBean.xml	チャンク別トランザクションモデル用の雛形
2	SingleTransactionBean.xml	全チャンク単一トランザクションモデル用の雛形
3	NoTransactionBean.xml	非トランザクションモデル用の雛形
4	ChunkTransactionForRestartBean.xml	チャンク別トランザクションモデル(リスタート機能有効)用の雛形
5	ChunkTransactionForControlBreakBean.xml	チャンク別トランザクションモデル(コントロールブレイク)用の雛形
6	PartitionChunkTransactionBean.xml	チャンク別トランザクションモデル(分割ジョブ)用の雛形
7	PartitionSingleTransactionBean.xml	全チャンク単一トランザクションモデル(分割ジョブ)用の雛形
8	PartitionNoTransactionBean.xml	非トランザクションモデル(分割ジョブ)用の雛形
9	PartitionChunkTransactionForRestartBean.xml	チャンク別トランザクションモデル(分割ジョブ)(リスタート機能有効)用の雛形
10	SequentialSingleTransactionBean.xml	分割ジョブにおいて、子ジョブを多重度1で逐次実行するための雛形

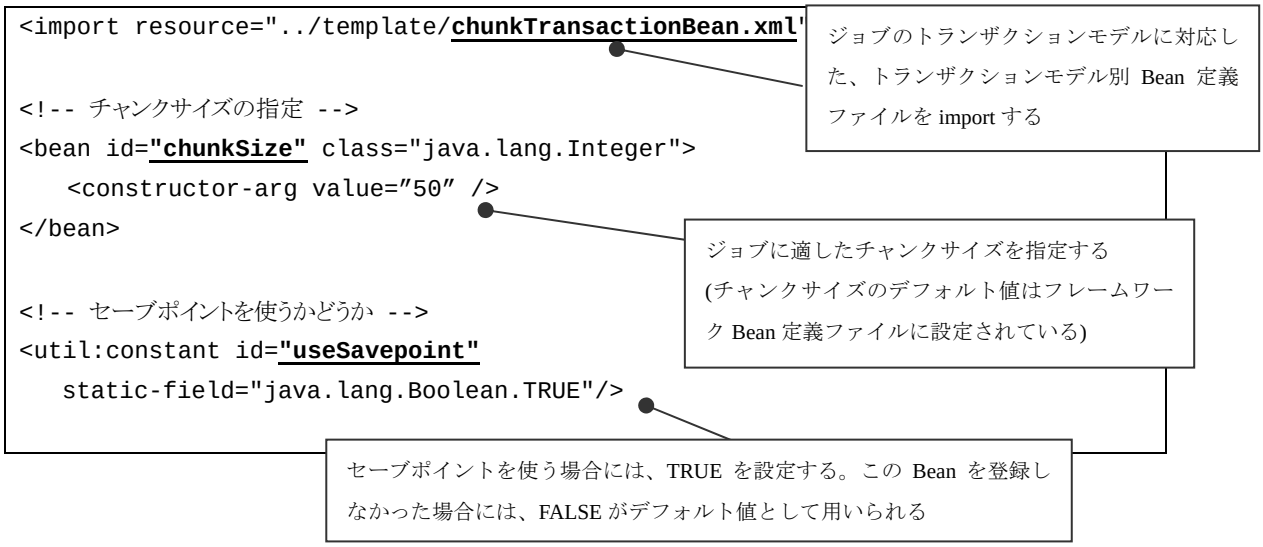
トランザクションモデル別 Bean 定義ファイル（チャンク別トランザクションモデル関連設定部分）

```
<!-- ====================== マネージャ定義 ====================== -->
<bean id="jobManager" parent="baseManager">
       . . .
   <property name="workQueueFactory">
     <bean class="jp.terasoluna.fw.batch.standard.StandardWorkQueueFactory">
        <property name="queueProcessor" ref="chunkTransactionQueueProcessor" />
     </bean>
   </property>
       . . .
</bean>

<!-- ====================== キュープロセッサ定義 ====================== -->
<bean id="chunkTransactionQueueProcessor" parent="baseQueueProcessor">
   <property name="worker" ref="chunkTransactionWorker" />
       . . .
</bean>

<!-- ====================== ワーカ定義 ====================== -->
<bean id="chunkTransactionWorker"
class="jp.terasoluna.fw.batch.springsupport.transaction.TransactionalWorker">
  <property name="jobWorker" ref="jobWorker" />
       . . .
</bean>

<!-- ====================== ジョブステータス定義 ====================== -->
<bean id="jobStatus"
class="jp.terasoluna.fw.batch.springsupport.transaction.TransactionalJobStatus">
  <property name="transactionManager" ref="transactionManager" />
       . . .
</bean>
```

トランザクションが設定された Worker が定義されている

P12 にて設定している DataSourceTransactionManager をプロパティとして設定している

➢ ジョブ Bean 定義ファイルの実装例



◆ 拡張ポイント

- トランザクションモデル別 Bean 定義の修正
  システムの特性にあわせて、フレームワーク Bean 定義ファイルやトランザクションモデル別 Bean 定義ファイルの定義を修正する。
  - デフォルトのチャンクサイズの修正
  - 分割ジョブでのデフォルトの多重度の修正
  - Spring トランザクション設定（タイムアウト等）の修正

- トランザクションモデル別 Bean 定義ファイル雛形の追加
  システムの特性に合わせて、プロジェクトでトランザクションモデル別 Bean 定義ファイル雛形を追加する。
  - 前処理／後処理が一定のパターンである場合に、それらの処理があらかじめ設定してあるトランザクションモデル別 Bean 定義ファイル雛形を追加
  - 特定の対象データを入力とするジョブが多数存在する場合には、その Collector があらかじめ設定してあるトランザクションモデル別 Bean 定義ファイル雛形を追加

- ビジネスロジックでのプログラマティックなトランザクション管理
  ビジネスロジック等のアプリケーションの Bean に、フレームワーク Bean 定義ファイルに記述されているトランザクションマネージャを DI することで、アプリケーションでプログラマティックなトランザクション制御を行うことができる。トランザクションマネージャからは、新規トランザクションや、現在アクティブなトランザクションを取得することができる。取得したトランザクションを使ってビジネスロジックでコミット／ロールバックを行うことができる。
  ビジネスロジックでのプログラマティックなトランザクション管理を行う場合には、フレームワークによるトランザクション管理と不整合を起こさないように、トランザクションモデルとして「非トランザクションモデル」を選択すること。
  ビジネスロジックでのプログラマティックなトランザクション管理は、まったく利用しないか、あるいは、特殊なトランザクション制御フローが必要なごく一部のジョブにのみ適用することが望ましい。特殊なトランザクション制御フローが必要なければ、トランザクション単位とメソッドの対応をとり、ソースコードの可読性、保守性の確保することを優先すること。
  以下に、TransactionProxyFactoryBean と BeanNameAutoProxyCreator での定義例を示す。TransactionProxyFactoryBean と BeanNameAutoProxyCreator の詳細は SpringAPI を参照のこと。

➢ ジョブ Bean 定義ファイル例（TransactionProxyFactoryBean を使用）

```
...
  <!-- BLogic の設定 -->
  <bean id="blogic"
    class="jp.co.nttdata.sample.blogic.SampleUpdateAgeBLogic">
    <property name="insertMemberService" ref="txInsertMemberService" />
  </bean>

  <!-- 共通処理にトランザクション定義を設定 -->
  <bean id="txInsertMemberService" parent="baseTransactionProxy">
    <property name="target" ref="insertMemberService" />
    <property name="transactionAttributes">
      <props>
        <prop key="insert*">PROPAGATION_REQUIRES_NEW,-java.lang.Exception</prop>
        <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
      </props>
    </property>
  </bean>

  <!-- 共通処理定義 -->
  <bean name="insertMemberService"
    class="jp.co.nttdata.functiontest.sample.service.SampleInsertMemberService">
    <property name="updateDAO" ref="updateDAO" />
  </bean>

</beans>
```

dataAccessContext-batch.xml に
定義されている baseTransactionProxy を指定する

トランザクションの対象 Bean を指定

➢ 共通処理

```
public class SampleInsertMemberService implements InsertMemberService {
...
   public void insertMember(Member member) {
      updateDAO.execute(SQLID, member);
   }
...
}
```

➢ ジョブ Bean 定義ファイル例（BeanNameAutoProxyCreator を使用）

```
・・・
  <!-- BLogic の設定 -->
  <bean id="blogic" class="jp.co.nttdata.sample.blogic.UpdateAgeBLogic">
<property name="insertMemberService" ref="insertMemberService" />
  </bean>
  <!-- トランザクション定義情報 -->
  <bean id="attrSource"
   class="org.springframework.transaction.interceptor.
                                        NameMatchTransactionAttributeSource">
    <property name="properties">
      <props>
        <prop key="insert*">
             PROPAGATION_REQUIRES_NEW,-java.lang.Exception
        </prop>
        <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
      </props>
    </property>
  </bean>

  <!-- トランザクション制御のメソッドインタセプタ(インタセプタの定義) -->
  <bean id="transactionInterceptor"
class="org.springframework.transaction.interceptor.TransactionInterceptor">
    <property name="transactionManager">
       <ref bean="transactionManager"/>
    </property>
    <property name="transactionAttributeSource">
       <ref local="attrSource"/>
    </property>
  </bean>

  <bean id="autoProxy"
class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="interceptorNames">
      <list><idref local="transactionInterceptor"/></list>
    </property>
    <property name="beanNames">
      <list><value>*Service</value></list>
    </property>
  </bean>
</beans>
```

トランザクションの対象 Bean を指定

関連機能

- 『BB-01 データベースアクセス機能』
- 『BD-01 ビジネスロジック実行機能』
- 『BH-01 例外ハンドリング機能』
- 『CA-01 トランザクション管理機能』

■ 使用例

- 機能網羅サンプル(functionsample-batch)
- チュートリアル(tutorial-batch)

■ 備考

- なし。

BB-01 データベースアクセス機能

■ 概要

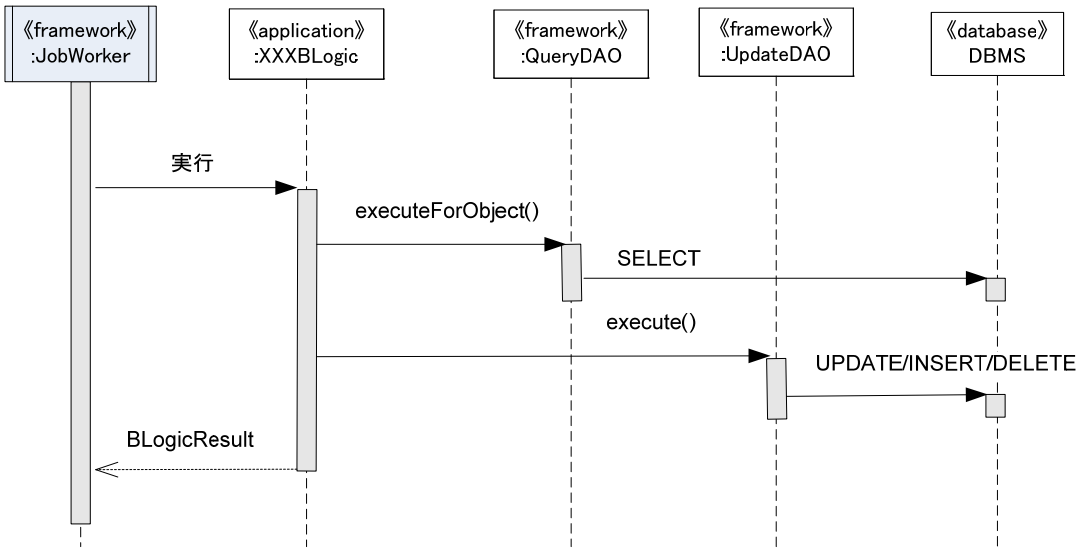
◆ 機能概要

- データベースアクセスは、共通機能の『CB-01 データベースアクセス機能』で提供される DAO を用いて行う。
  - DAO インタフェースにより、JDBC API および RDBMS 製品や O/R マッピングツールに依存する処理をビジネスロジックから分離する。
  - DAO インタフェースのデフォルト実装として Spring ＋ iBATIS 連携機能を利用した DAO 実装を提供する。
  - フレームワークが Spring framework の機能を使用したトランザクション制御を行うため、ビジネスロジック実装者がコネクションオブジェクトの受け渡しや、トランザクションを考慮した処理を実装する必要がない。
  - iBATIS の詳細な使用方法や設定方法などは、iBATIS のリファレンスを参照すること。
  - SQL 文は、iBATIS の仕様にしたがって、設定ファイルに記述する。

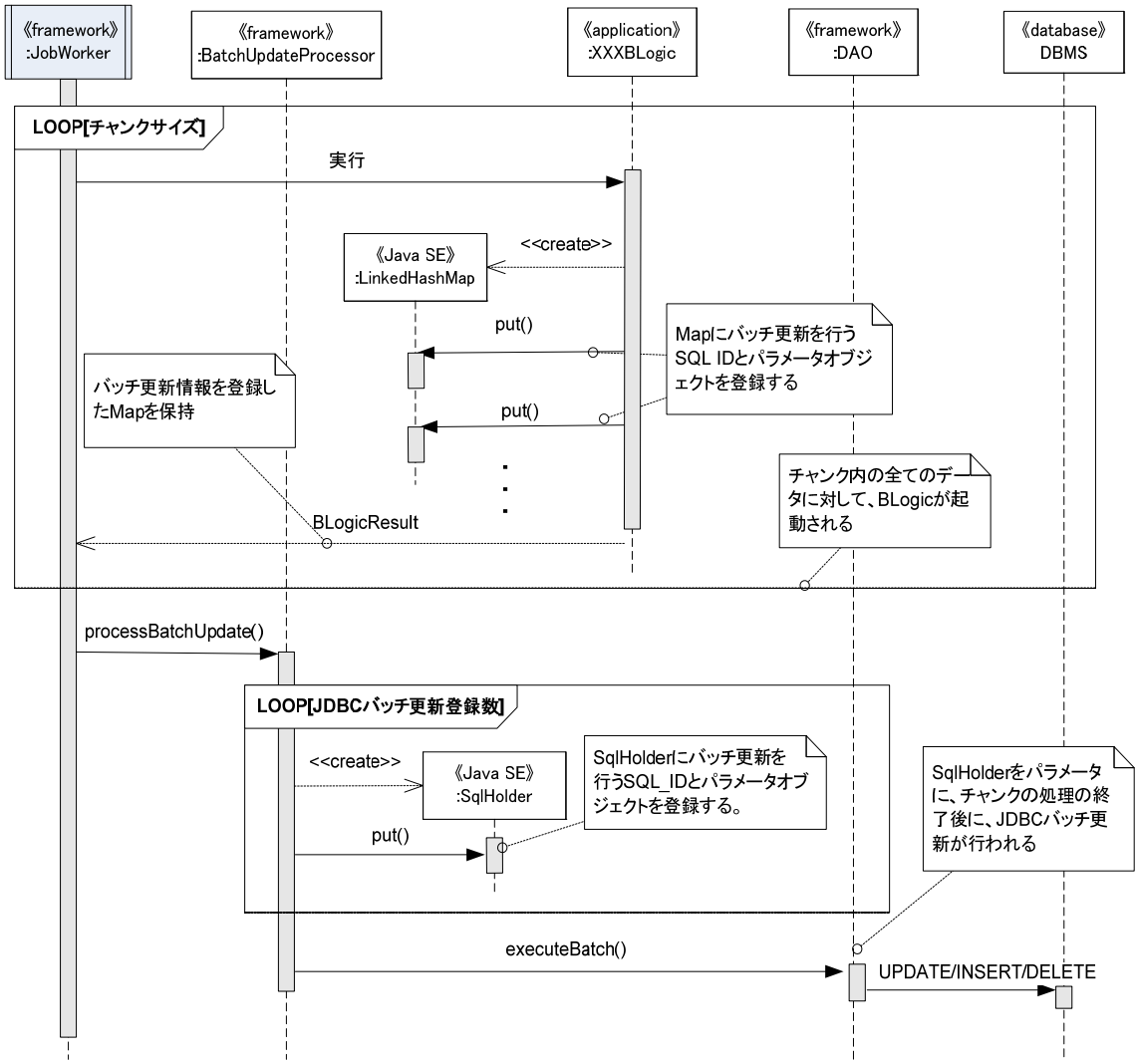
- データベースアクセスは、複数のビジネスロジックにまたがった JDBC バッチ更新機能を提供する。
  - java.sql.PreparedStatement クラスの addBatch()メソッド、executeBatch()メソッドによる処理を本説明書では JDBC バッチ更新と呼ぶ。
  - java.sql.PreparedStatement クラスの addBatch()メソッド、executeBatch()メソッドによる処理については、JDK の API ドキュメントを参照のこと。
  - ビジネスロジックでは、JDBC バッチ更新を行う更新 SQL ID と、そのパラメータオブジェクトを BLogicResult に登録してフレームワークにリターンする。
  - フレームワークは、BLogicResult に登録された更新 SQL ID とそのパラメータオブジェクトを受け取り、チャンクに含まれていた対象データに対するビジネスロジックの処理が終了した後、JDBC バッチ更新を行う。

◆ 概念図

- DAO によるデータベースアクセス処理
  ビジネスロジックでは、フレームワークが提供する DAO によってデータベースアクセスを行うことができる。

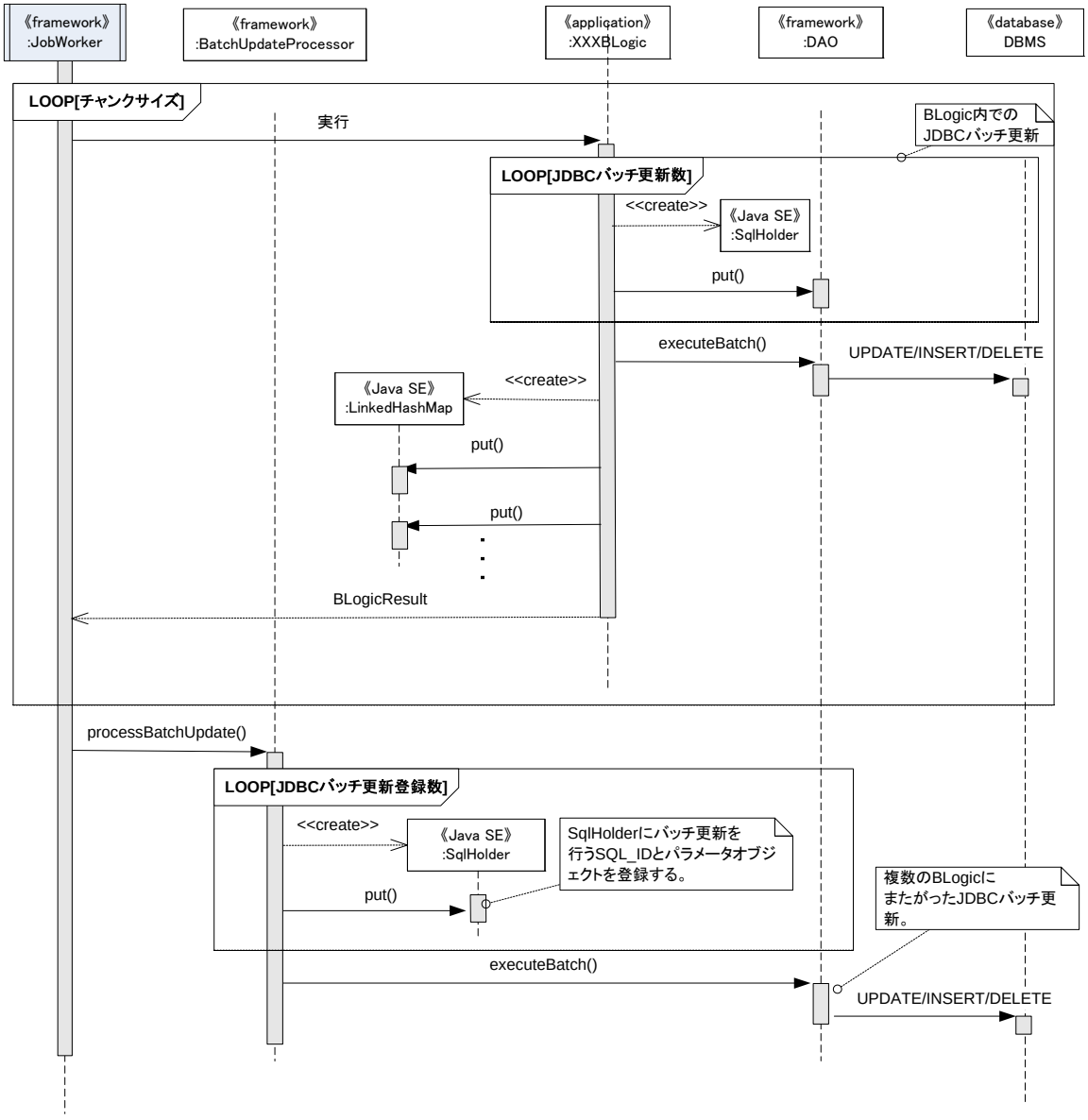


- 複数のビジネスロジックにまたがったJDBCバッチ更新
  チャンク内の全ての処理対象データに対して、ビジネスロジックを呼び出した後、フレームワークによってJDBCバッチ更新が行われる。
  フレームワークへ JDBC バッチ更新を依頼する場合には、BLogicResult オブジェクトに更新対象の SQL ID、およびパラメータオブジェクトを登録する。但し、一つのビジネスロジックで同じ SQL_ID の SQL を複数登録することは出来ない。



◆ 解説

- 「ビジネスロジック内の JDBC バッチ更新」と「ビジネスロジックをまたがった JDBC バッチ更新」
  TERASOLUNA-Spring の DAO では、JDBC バッチ更新の機能が提供されている。
  ビジネスロジック内部に閉じる JDBC バッチ更新は、TERASOLUNA-Spring の DAO の JDBC バッチ更新機能を使って行う。
  BLogicResult に登録して行うのは、複数のビジネスロジックにまたがって JDBC バッチ更新を行う場合である。
  「ビジネスロジック内の JDBC バッチ更新」と「ビジネスロジックをまたがった JDBC バッチ更新」は、一つのジョブで両方を使うことができる。



- 「ビジネスロジックをまたがった JDBC バッチ更新」の SQL 発行タイミング
  ビジネスロジックをまたがった JDBC バッチ更新では、BLogicResult に登録した時点では SQL は発行されていないことに留意すること。ビジネスロジック内の後続する処理や、後続のビジネスロジックが、当該 SQL の更新結果に依存するような場合には「ビジネスロジックをまたがった JDBC バッチ更新」を使うことができない。
- 「ビジネスロジックをまたがった JDBC バッチ更新」の SQL 実行順番
  ビジネスロジックをまたがった JDBC バッチ更新では、BLogicResult に登録した SQL の順番を保障しない。JDBC の API では、「バッチに追加された順序で逐次実行される」ことになっているが、フレームワークでは性能向上のために SQL ID の昇順にソートしてから実行している。したがって、ビジネスロジックをまたがった JDBC バッチ更新では、順序性を持たない SQL のみ登録すること。。

- 「ビジネスロジックをまたがった JDBC バッチ更新」の更新結果の判定
  JDBC の API では、JDBC バッチ更新の結果は「バッチ内に含まれているそれぞれの更新 SQL によって更新された行数の配列」が返されることになっている。しかし、Oracle 10g、PostgreSQL 8 では DBMS（あるいは JDBC ドライバ）の制限から、JDBC バッチ更新を行った際の更新行数を取得することができない。
  したがって、JDBC バッチ更新を採用できる更新処理は、以下のようにそれぞれの更新処理の件数を判定する必要がないもの（失敗した場合には例外がスローされるもの）に限ること。
  - 単純な一件の INSERT
  - 登録済みであることがわかっている（=空振りしない）主キーによる UPDATE、DELETE

  「ビジネスロジックをまたがった JDBC バッチ更新」で例外が発生した場合の処理については、『BH-01 例外ハンドリング機能』を参照のこと。

- データベースアクセス時の例外
  フレームワークから提供される DAO インタフェースのデフォルト実装を利用した場合には、データベースアクセス時に発生した例外は、Spring Framework の実行時例外としてスローされる。
  主キー重複や、ロック獲得のタイムアウトなどの例外に対して、業務ロジックで回復処理を行う場合には、Spring Framework の例外をキャッチする。
  以下に Spring Framework がスローするデータアクセス例外のうち、AP で回復の可能性があるものを示す。

項番	Spring 例外クラス	概要
1	`org.springframework.dao.DataIntegrityViolationException`	主キーが重複してしまうような SQL を発行した場合に、スローされる例外。
2	`org.springframework.dao.CannotAquireLockException`	“SELECT FOR UPDATE”でロックが獲得できなかった場合などにスローされる例外。

  上記の例外クラスは、代表的な例外クラスである。詳細は Spring Framework の API ドキュメントを参照すること。

  Spring Framework には、SQL 例外トランスレータという概念があり、SQL 例外をエラーコード等に応じて別の例外に変換するための仕組みが用意されている。SQL 例外トランスレータはデフォルトで用意されているが、拡張を行うこともできる。詳細は、Spring Framework の SQLExceptionTranslator インタフェースなどを参照のこと。

- SQL 文への SQL ID の埋め込み
  iBATIS の設定ファイルに記述する SQL 文には、SQL ID をコメントで埋め込むことを推奨する。
  - SQL 文に、SQL ID を埋め込むことで、DBMS 付属のツール等で負荷の高い SQL をリストアップしたときに、AP のどの部分から発行している SQL かを特定しやすくなる。

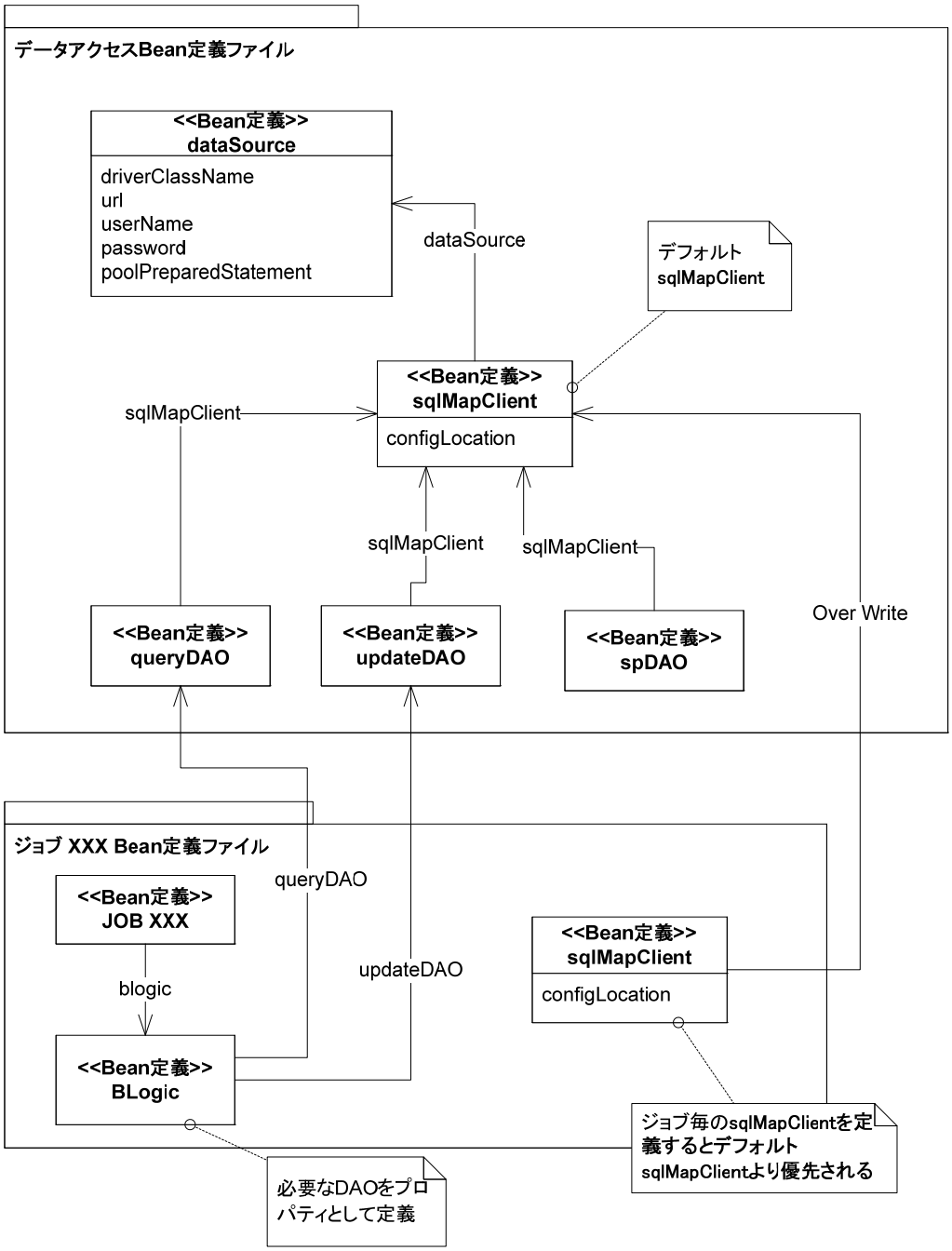
- SQL 発行ログ
  Spring＋iBATIS のデフォルトの DAO 実装を用いる場合には、iBATIS の機能により SQL 発行ログを出力することができる。
  - 出力内容の詳細などは、`com.ibatis.common.jdbc.logging` パッケージのクラス群の API ドキュメント、実装を参照のこと。

- リソースのキャッシュ
  コネクションプーリング、各種キャッシュ機能は、フレームワークがライブラリを使うことで提供する。
  - コネクションプーリングは、Jakarta Commons DBCP を利用する。
  - PreparedStatement のキャッシュは、DBCP の機能を利用する。
  - マスタ類の検索結果のキャッシュは、iBATIS の機能を利用する。

- データベースコネクション数
  通常ジョブでは、データベースコネクションは Collector で使われるもの、および Worker で使われるもの、の 2 本が利用される。
  また、分割ジョブでは親ジョブの Collector で使われるもの、および子ジョブの Collector、Worker で使われるものがあるため、以下の本数分のコネクションが利用される。
  分割ジョブでのコネクション数：
  1 [親ジョブの Collector]＋（ **2** [子ジョブの Collector、Worker]×**多重度**）

  なお、通常ジョブの場合であっても、分割ジョブの場合であっても、ファイルアクセスのみなどでデータベースアクセスを行わないジョブである場合には、コネクションは利用されない。

- データアクセス Bean 定義ファイル、ジョブ Bean 定義ファイルの構造
  データアクセス Bean 定義ファイルには、フレームワークの DAO が設定されている。
  ジョブ Bean 定義ファイルには、ビジネスロジックで必要な DAO をプロパティに定義し、フレームワークの DAO をセットすること。



■ 使用方法

◆ コーディングポイント

- データソースの Bean 定義
  データソースの設定は、データアクセス Bean 定義ファイルに定義する。
  環境に依存する DB 接続のための項目は、プロパティファイルに外だしすることが望ましい。プロパティファイルに外だしする場合には、
  `PropertyPlaceholderConfigurer` を定義する。

  ➢ データアクセス Bean 定義ファイルの例

```
<!-- 外だしのプロパティファイルの読込み設定。 -->
<!-- プレースホルダ -->
<import resource="classpath:common/PlaceHolderConfig.xml" />
<!-- DataSourceの設定。 -->
<!--
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
   <property name="jndiName" value="TerasolunaDataSource" />
   <property name="resourceRef" value="false" />
</bean>
-->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close" >
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
    <property name="poolPreparedStatement" value="true"/>
</bean>
```

AP サーバの DataSource 使用時の設定

DBCP により PreparedStatement をキャッシュすることができる

  ➢ プロパティファイルの例（jdbc.properties）

```
jdbc.driverClassName=oracle.jdbc.OracleDriver
```

  ➢ 業務ロジックの例

```
LinkedHashMap<String, Object> batchUpdateMap = new LinkedHashMap<String, Object>();
...
batchUpdateMap.put("insertSeikyu", seikyu);
...
return new BLogicResult(ReturnCode.NORMAL_CONTINUE, batchUpdateMap);
```

バッチ更新情報を格納するマップを作成する

JDBC バッチ更新の SQL とパラメータを登録する

BLogicResut 生成時にバッチ更新情報を格納したマップを渡す

- iBATIS マッピング定義ファイル
  このファイルは、ビジネスロジックで利用する SQL 文と、その SQL の実行結果を JavaBean にマッピングするための定義を設定する。ジョブ単位にファイルを作成することが望ましい。SQL の詳細な記述方法に関しては、iBATIS のリファレンスを参照のこと。
  - Select 文の実行例
    - Select 文の実行には、<select>要素を使用する。
    - resultClass 属性に SQL の結果を格納するクラスを指定する。結果が複数の場合は、指定したクラスの配列が返却される。

```
<select id="getUser"
        resultClass="jp.terasoluna.·····.service.bean.UserBean">
   /* ID=getUser FILE=login-sql.xml */
   SELECT ID, NAME, AGE, BIRTH FROM USERLIST WHERE ID = #ID#
</select>
```

SQL 文に SQL ID を埋め込むことを推奨する。

  - Insert、Update、Delete 文の実行
    - Insert 文の実行には、<insert>要素を使用する。
    - Update 文の実行には、<update>要素を使用する。
    - Delete 文の実行には、<delete>要素を使用する。
    - parameterClass 属性に、登録するデータを保持しているクラスを指定する。parameterClass 属性に指定したクラス内のプロパティ名の前後に「#」を記述した部分に、値が設定された SQL が実行させる。

```
<insert id="insert_User"
  parameterClass="jp.terasoluna.·····.service.bean.UserBean">
   INSERT INTO USERLIST (ID, NAME, AGE, BIRTH ) VALUES (
   #id#, #name#, #age#, #birth#)
</insert>
```

  - Procedure 文の実行
    - Procedure 文の実行には、< procedure>要素を使用する。
    - 基本的な使用方法は、他の要素と同様だが、<select>要素より、属性が少なくなっている。
    - 値の設定、結果の取得は他の要素と異なり、parameterMap 属性、および<parameterMap>要素を使用する。

```
<sqlMap namespace="user">
   <parameterMap id="UserBean" class="java.util.HashMap">
      <parameter property="inputId" jdbcType="NUMBER"
               javaType="java.lang.String" mode="IN"/>
      <parameter property="name" jdbcType="VARCHAR"
               javaType="java.lang.String" mode="OUT"/>
   </parameterMap>
   <procedure id="selectUserName" parameterMap="user.UserBean">
     {call SELECTUSERNAME(?,?)}
   </procedure>
```

- iBATIS 設定ファイル
  iBATIS 設定ファイルには、iBATIS マッピング定義ファイルの場所の指定のみ記述する。データソース、トランザクション管理は、Spring との連携機能を利用して行うため、本ファイルでその設定はしないこと。
  - <sqlMap>要素は複数記述することができるため、iBATIS マッピング定義ファイルを分割した際は、複数記述すること。

```
<sqlMapConfig>
  <sqlMap resource="sqlMap.xml"/>
  <sqlMap resource="registerSqlMap.xml"/>
</sqlMapConfig>
```

- SqlMapClientFactoryBean の Bean 定義
  Spring で iBATIS を使用する場合、SqlMapClientFactoryBean を使用して iBATIS 設定ファイルの Bean 定義を DAO に設定する必要がある。SqlMapClientFactoryBean は、iBATIS のデータアクセス時に利用されるメインのクラス「SqlMapClient」を管理する役割を持つ。
  同期型でジョブを起動する場合には、iBATIS 設定ファイルはジョブ毎に定義することが出来る。iBATIS 設定ファイルをジョブ毎に設定することで、当該ジョブに必要な SQL のみを読み込むことができる。
  - dataAccessContext-batch.xml（SqlMapClientFactoryBean の Bean 定義）

```
<bean id="sqlMapClient"
   class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="configLocation" ref="sqlMapConfigFileName"/>
</bean>
```

  - DefaultValueBean.xml の設定（iBATIS 設定ファイル名のデフォルト設定）
    DefaultValueBean.xml には、iBATIS 設定ファイルのデフォルト定義が設定されている。ジョブ Bean 定義に sqlMapConfigFileName の Bean 定義が無い場合はデフォルト定義が参照される。

```
<bean id="sqlMapConfigFileName" class="java.lang.String">
       <constructor-arg value=”common/sql-map-config.xml” />
</bean>
```

  - ジョブ Bean 定義の設定（iBATIS 設定ファイル名の設定）
    ジョブ毎に指定した iBATIS 設定ファイルを参照する場合には、ジョブ Bean 定義ファイルに sqlMapConfigFileName の Bean 定義を行う。非同期型でジョブを起動した場合には、この設定は無視される。

```
<bean id="sqlMapConfigFileName" class="java.lang.String">
       <constructor-arg value=”UC001/uc001-sql-map-config.xml” />
</bean>
```

- DAO の Bean 定義
  - DAO 実装クラスは、基本的にデータアクセス Bean 定義ファイルに定義する。また、DAO もトランザクション設定対象の Bean である。トランザクションの設定方法は『BA-01 トランザクション管理機能』のフレームワーク説明書を参照のこと。
    また、Bean 定義時に DAO 実装クラスの“sqlMapClient”プロパティに iBATIS 設定ファイルの Bean 定義を設定する必要がある。

```
<bean id="sqlMapClient"
     class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
  <property name="configLocation" ref="sqlMapConfigFileName"/>
  <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="queryDAO"
     class="jp.terasoluna.fw.dao.ibatis.QueryDAOiBatisImpl">
     <property name="sqlMapClient" ref="sqlMapClient"/>
</bean>
```

- QueryDAOiBatisImpl のメソッドの戻り値の指定（『CB-01 データベースアクセス機能』の再掲）
  - QueryDAOiBatisImpl の以下のメソッドを使用する場合は、戻り値の型と同じ型のクラスを引数に渡す必要がある。
    これにより、ビジネスロジックでのクラスキャストエラーの発生を避けることができる（DAO が IllegalClassTypeException をスローする）。
    - ✧ executeForObject (String sqlID, Object bindParams, Class clazz)
    - ✧ executeForObjectArray(String sqlID, Object bindParams, Class clazz)
    - ✧ executeForObjectArray(String sqlID, Object bindParams, Class clazz, int beginIndex, int maxCount)

```
UserBean bean = dao.executeForObject("getUser", null, UserBean.class);
```

- UpdateDAOiBatisImpl を使用したデータ登録例（『CB-01 データベースアクセス機能』の再掲）

  UpdateDAOiBatisImpl を使用して、データベースに情報を登録する場合の設定およびコーディング例を以下に記述する。Bean 定義ファイルの定義・設定方法は、QueryDAOiBatisImpl と同様なため省略する。

  - ビジネスロジック

    Bean 定義ファイルにて設定された DAO のメソッドを使用する。

```
private UpdateDAO updateDAO = null;
    ......Setterは省略
public BLogicResult execute(...) {
    ......
    updateDAO.execute("insertUser", bean);
    ......
}
```

設定された DAO を使用して、データベースにデータを登録する。

- UpdateDAOiBatisImpl を使用した複数データの登録例（ビジネスロジック内の JDBC バッチ更新処理）

  UpdateDAOiBatisImpl を使用して、データベースに複数の情報を登録する場合の設定およびコーディング例を以下に記述する。Bean 定義ファイルの定義・設定方法は、QueryDAOiBatisImpl と同様なため省略する。

  詳細は UpdateDAOiBatisImpl の JavaDoc を参照のこと。

  - ビジネスロジック

    Bean 定義ファイルにて設定された DAO の executeBatch(List<SqlHolder>)メソッドを使用する。

```
UserBean[] bean = map.get("userBeans");
   List<SqlHolder> sqlHolders = new ArrayList<SqlHolder>();
     for (int i = 0; i < bean.length; i++) {
        sqlHolders.add(new SqlHolder("insertUser", bean[i]);
     }
     updateDAO.executeBatch(sqlHolders);
     ......
```

更新対象の sqlId、パラメータとなるオブジェクトを保持した SqlHolder のリストを作成する。

- StoredProcedureDAOiBatisImpl を使用したデータ取得例（『CB-01 データベースアクセス機能』の再掲）

  StoredProcedureDAOiBatisImpl を使用して、データベースから情報を取得する場合の設定およびコーディング例を以下に記述する。Bean 定義ファイルの定義・設定方法は、QueryDAOiBatisImpl と同様なため省略する。

  - ビジネスロジック

    Bean 定義ファイルにて設定された DAO のメソッドを使用する。

```
private StoredProcedureDAO spDAO = null;
   ......Setterは省略
public BLogicResult execute(...) {
   ......
   Map<String, String> params = new HashMap<String, String>();
   params.put("inputId", bean.getId());
   spDAO.executeForObject("selectUserName", params);
   ......
}
```

設定された DAO を使用してプロシージャーを実行する。

  - iBATIS マッピング定義ファイル
    - ✧ プロシージャーの入出力を格納するための設定を<parameterMap>要素にて記述する。jdbcType 属性を指定すること。詳細な設定方法は、iBATIS のリファレンスを参照のこと。

```
<sqlMap namespace="user">
   <parameterMap id="UserBean" class="java.util.HashMap">
      <parameter property="inputId" jdbcType="NUMBER"
               javaType="java.lang.String" mode="IN"/>
      <parameter property="name" jdbcType="VARCHAR"
               javaType="java.lang.String" mode="OUT"/>
   </parameterMap>
   <procedure id="selectUserName" parameterMap="user.UserBean">
     {call SELECTUSERNAME(?,?)}
   </procedure>
```

  - 実行するプロシージャー（Oracle を利用した例）

```
CREATE OR REPLACE PROCEDURE SELECTUSERNAME
(inputId IN NUMBER, name out VARCHAR2) IS
BEGIN
   SELECT name INTO name FROM userList WHERE id = inputId ;
END ;
```

◆ 拡張ポイント

- フレーワーク DAO 実装の拡張
  フレームワークから提供されている DAO 実装はデフォルト実装であり、プロジェクトにあわせて例外処理などを拡張することができる。
  たとえば、解析を容易にするため、データアクセス例外が発生した場合に、実行した SQL ID とバインドパラメータのダンプをログに出力する、例外オブジェクトに載せるという処理がしばしば行われる。このような処理は、フレームワークのデフォルト実装を拡張することで実装できる。

- データベース接続パスワードの暗号化
  Spring フレームワークの PropertyPlaceholderConfigurator を拡張することで、データベースへの接続パスワードを暗号化することができる。
  プロジェクトの要件に応じて、暗号の強度や秘密鍵の管理方法を検討して拡張を行うことができる。

■ 関連機能

- 『BA-01 トランザクション管理機能』
- 『BD-01 ビジネスロジック実行機能』
- 『BE-05 処理結果ハンドリング機能』
- 『BH-01 例外ハンドリング機能』
- 共通機能『CB-01 データベースアクセス機能』

■ 使用例

- 機能網羅サンプル(functionsample-batch)
- チュートリアル(tutorial-batch)

■ 備考

- なし。

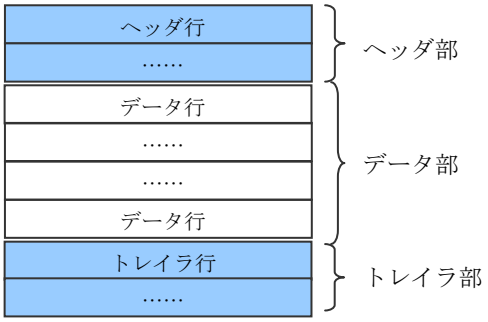
BC-01 ファイルアクセス機能

■ 概要

◆ 機能概要

- CSV 形式、固定長形式、可変長形式ファイルの入出力機能を提供する。
  - ファイル入力機能は順次読込のみ提供する。

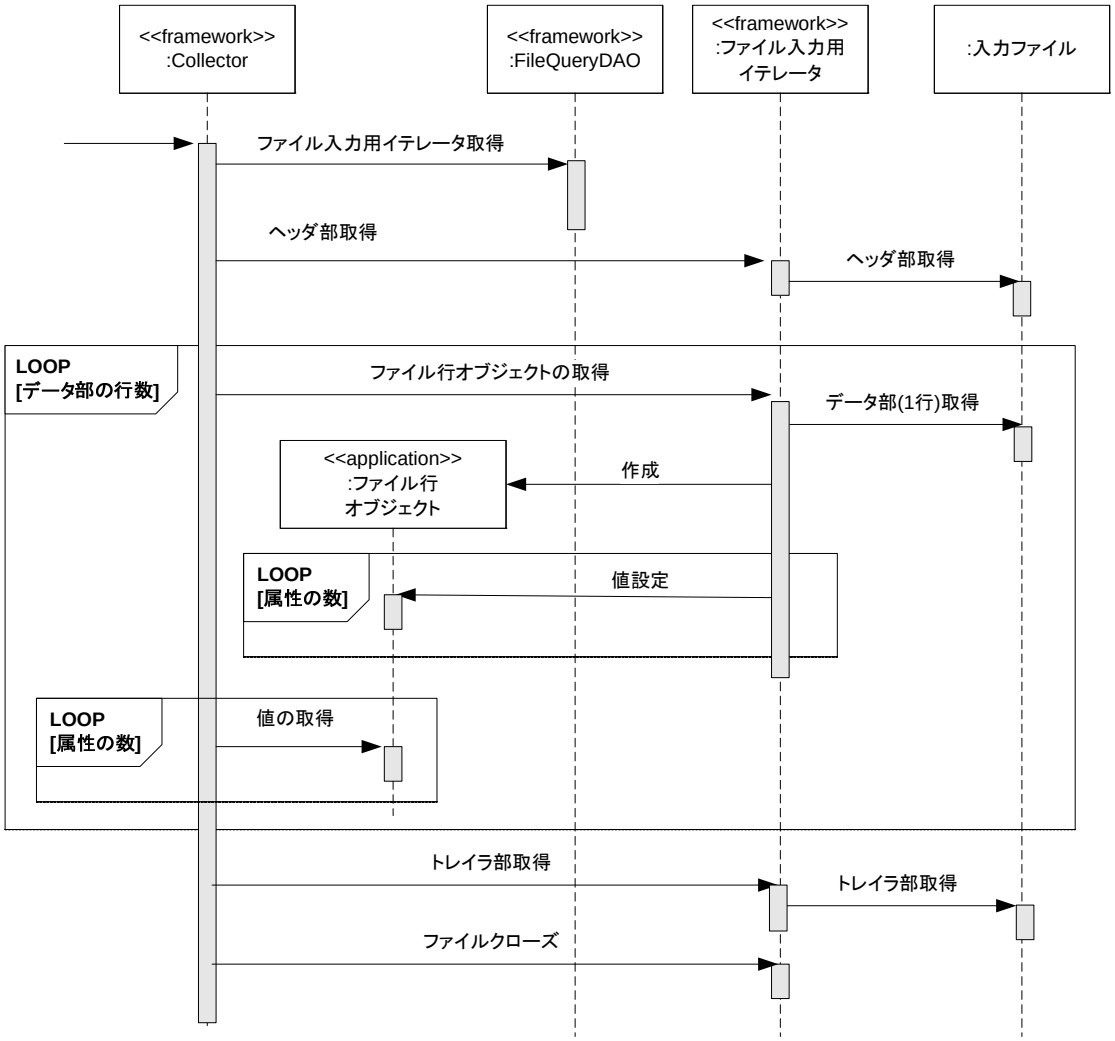
- ファイルアクセス機能で対象とするファイル構造は下図のとおりである。
  - ヘッダ部、トレイラ部の無いファイルについては、ヘッダ部、トレイラ部を 0 行とする。
  - データ部のデータ構造はファイル行オブジェクト（POJO）にアノテーションを使用して定義する。



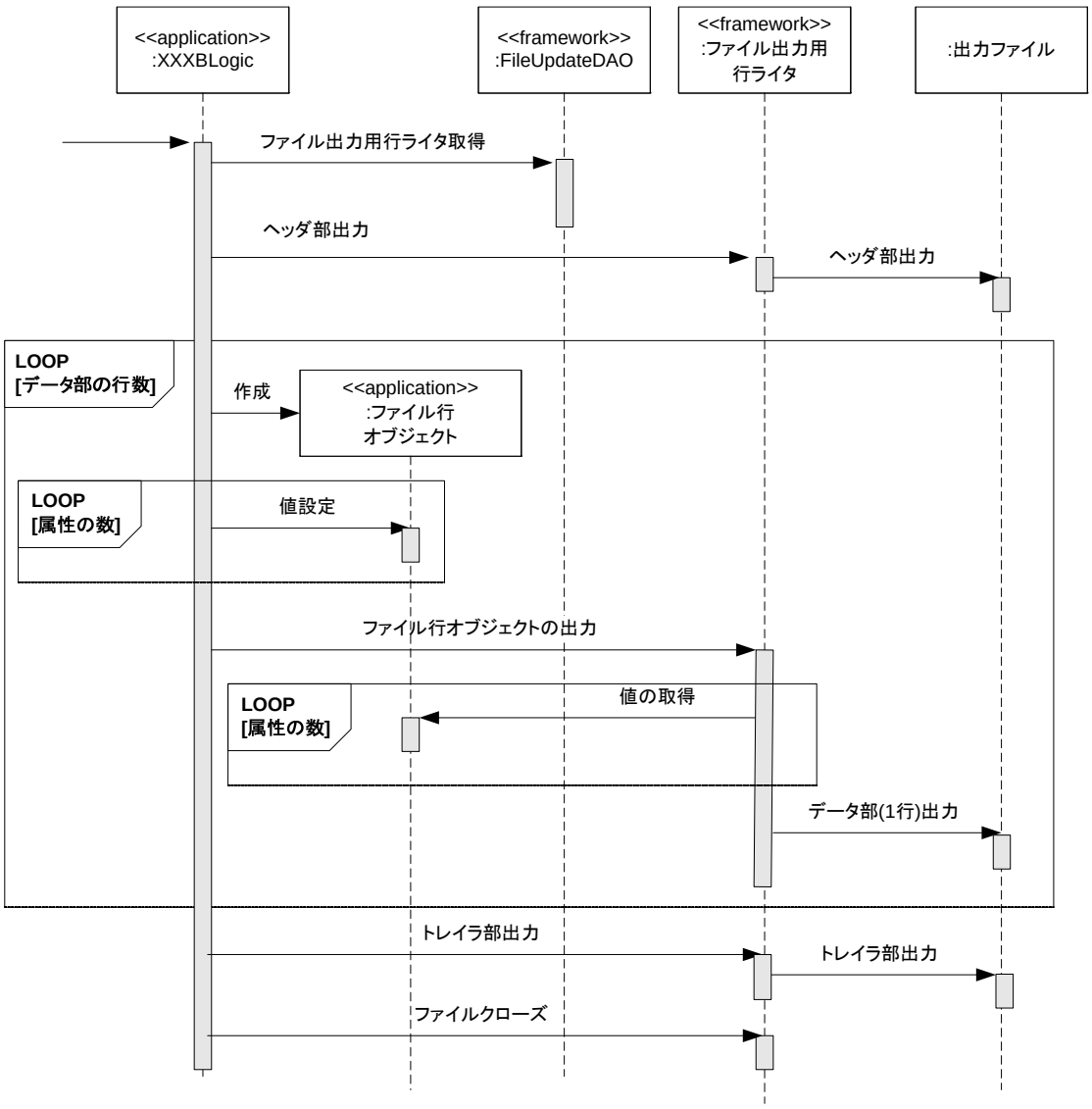
- 入出力データに対するフォーマット処理を行う。
  - ファイル行オブジェクト（POJO）の定義により、項目に対するパディング(Padding)、トリム（Trim）、文字変換（StringConverter）等が指定できる。

◆ 概念図

- ファイル入力処理の概念図（Collector から利用される場合）



- ファイル出力処理の概念図（ビジネスロジックから利用される場合）



◆ 解説

- ファイルアクセス機能で取り扱うファイル
  TERASOLUNA-Batch のファイルアクセス機能は CSV 形式、固定長形式、可変長形式のファイル入出力機能を提供する。ファイル内の各行の項目数及び項目の並び順は同一である必要がある。

  - CSV 形式
    CSV 形式とは、データを「,(カンマ)」で区切ったものである。データを区切る際に使用しているカンマを特に"区切り文字”と呼ぶ。CSV 形式は可変長ファイルの区切り文字を「,(カンマ)」に固定したものになる。

  - 固定長形式
    固定長形式とは、対象データの 1 行の各項目の長さ（バイト数）が全ての行で同じであるもの。対象データを区切る方法は、各項目に割り当てられているバイト数をもとに行う。

  - 可変長形式
    可変長形式とは、対象データの 1 行の各項目の長さ（バイト数）が可変である（異なる）もの。対象データは"区切り文字” を使って区切る。

- ファイル行オブジェクト
  ファイル入出力を使用する場合には、ファイルのデータ部の 1 行分のデータに対応するように Java Bean のクラスを作成する。作成する Java Bean のクラスを、本フレームワークではファイル行オブジェクトと呼ぶ。
  ファイル行オブジェクトのクラスには、ファイル全体に関わる定義情報（改行文字等）を Java アノテーションにより設定する。
  ファイル行オブジェクトのクラスが持つ属性には、ファイルの個々の項目の定義情報（バイト長等）を Java アノテーションにより設定する。

- ファイル全体に関わる定義情報（FileFormat アノテーション）
  ファイル全体に関わる定義情報は、ファイル行オブジェクトのクラスに対してアノテーション FileFormat により設定する。FileFormat アノテーションは入力ファイル、および出力ファイルのどちらの場合にも同じアノテーションを設定する。
  - アノテーションFileFormat設定項目

項番	論理項目名 物理項目名	説明	デフォルト値	CSV		固定長		可変長		その他	
				入	出	入	出	入	出	入	出
1	行区切り文字 lineFeedChar	行区切り文字(改行文字)を設定する。	システムデフォルト	○	○	○	○	○	○	○	○
2	区切り文字 delimiter	「,(カンマ)」等の区切り文字を設定する。	「,(カンマ)」	×	×	×	×	○	○		
3	囲み文字 encloseChar	「“(ダブルクォーテーション)」等のカラムの囲み文字を設定する。	なし	○	○	×	×	○	○		
4	ファイルエンコーディング fileEncoding	入出力を行うファイルのエンコーディングを設定する。	システムデフォルト	○	○	○	○	○	○	○	○
5	ヘッダ行数 headerLineCount	入力ファイルのヘッダ部に相当する行数を設定する。	0	○		○		○		○	
6	トレイラ行数 trailerLineCount	入力ファイルのトレイラ部に相当する行数を設定する。	0	○		○		○		○	
7	ファイル上書きフラグ overWriteFlg	出力ファイルと同じ名前のファイルが存在する場合に上書きするかどうかを設定する。 [true/false]（上書きする/上書きしない）	FALSE		○		○		○		○

※○の項目は必要に応じて設定可。×の項目は設定できないことを表している（×の項目を設定した場合、実行時にエラーとなる）。無印は設定を無視することを表している。
※「行区切り文字」の"システムデフォルト"とは、
System.getProperty(”line.separator”);で取得できる実行環境に依存した値である。
※「ファイルエンコーディング」の"システムデフォルト"とは、
System.getProperty(”file.encoding”);で取得できる実行環境に依存した値である。
※「行区切り文字」、「区切り文字」でタブ、改行文字を使用する場合、Java 言語仕様で定められているエスケープシーケンス（¥t、¥r 等）で記述すること。
※可変長の「区切り文字」として「￥u0000」を設定することはできない。
※「区切り文字」と「囲み文字」は同一の値を設定することができない。

- ファイル項目の定義情報
  （InputFileColumn アノテーション、OutputFileColumn アノテーション）
  ファイル項目の定義情報は、ファイル行オブジェクトのクラスが持つ属性に対してアノテーション InputFileColumn、OutputFileColumn により設定する。アノテーション InputFileColumn、OutputFileColumn は入力ファイル、および出力ファイルのどちらの場合にも同じアノテーションを設定する。
  ひとつのファイル行オブジェクトが入力ファイル、および出力ファイルの両方で使用される場合には、ひとつの属性に対して二つのアノテーション
  （InputFileColumn、OutputFileColum）を設定する。

  - InputFileColumn,OutputFileColumn の設定項目

項番	論理項目名 物理項目名	説明	デフォルト値	CSV		固定長		可変長	
				入	出	入	出	入	出
1	カラムインデックス columnIndex	データ部の 1 行のカラムの内、何番目のデータをファイル行オブジェクトの属性に格納するのかを設定する。インデックスは「0(ゼロ)」から始まる整数。	なし	◎	◎	◎	◎	◎	◎
2	フォーマット columnFormat	BigDecimal 型、Date 型に対するフォーマットを設定する。	なし	○	○	○	○	○	○
3	バイト長 bytes	各カラムに対するバイト長を設定する。	なし	○	○	◎	◎	○	○
4	パディング種別 paddingType	パディングの種別を設定する。列挙型 PaddingType から値を選択する。[RIGHT/LEFT/NONE] (右寄せ/左寄せ/パディングなし)	NONE		○		○		○
5	パディング文字 paddingChar	パディングする文字を設定する(半角文字のみ設定可能)。	なし		○		○		○
6	トリム種別 trimType	トリムの種別を設定する。列挙型 TrimType から値を選択する。[RIGHT/LEFT/BOTH /NONE] (右寄せ/左寄せ/両側/トリムなし)	NONE	○	○	○	○	○	○
7	トリム文字 trimChar	トリムする文字を設定する(半角文字のみ設定可能)。	なし	○	○	○	○	○	○
8	文字変換種別 stringConverter	String 型のカラムについて、大文字変換等を設定する。StringConverter インタフェースの実装クラスを指定する。 StringConverterToUpperCase.class (大文字に変換) / StringConverterToLowerCase.class(小文字に変換) / NullStringConverter.class(変換しない)	NullStringConverter.class	○	○	○	○	○	○

※ ◎の項目はアノテーションを設定する際の必須項目(必須項目を設定しなかった場合、実行時にエラーとなる)。○の項目は必要に応じて設定可。無印は設定を行っても有効にならないことを表している。

※ バイト長とは、入力時はファイルから取得する時点の長さであり、各種変換処理後の長さとは異なる。出力時はファイルへ出力する時点の長さであり、各種

変換処理後の長さである。

※ パディング種別、トリム種別を指定したときには、それぞれパディング文字、トリム文字を必ず設定すること。

※ パディング種別で NONE 以外を指定したときはバイト長を必ず設定すること。ここでのバイト長は、パディング処理を行った後のバイト長を設定すること。

※ 入力処理時にパディングを設定した場合、取得データがバイト長で設定した長さ以外の場合はバイト長チェックでエラーが発生し、バイト長で設定した長さと一致する場合はパディングすべきデータ数が 0 となるため、パディング処理を行っても取得データと同じになる。つまり、入力時にパディングの設定を行っても有効にならないことに留意すること。

※ 変換処理の順番は、入力時と出力時で異なることに留意すること。

- 入力時はバイト数チェック、トリム処理、パディング処理、文字変換処理である。
- 出力時はトリム処理、パディング処理、文字変換処理、バイト数チェックである。

- データ項目定義と異なるデータを入出力した場合の例外処理
  対象データの入力の際、アノテーションの記述と異なるデータがあった場合、フレームワークは例外を発生させる。例外が発生する例としては、日付型のフォーマットを設定しているところに、数値型のデータを格納しようとした場合などが挙げられる。

- ファイル行オブジェクトの属性について
  - ファイル行オブジェクトで使用できる属性の型は、java.lang.String、int 、java.math.BigDecimal、java.util.Date の 4 種類とする。
  - 各属性には値を操作するために可視性が public である setter/getter を用意すること。

- フォーマットについての補足
  数値型、日付型の入出力は、フォーマットとして入力した文字列に沿ってデータの入出力を行う。詳細については、Java Platform Standard Edition (Java SE) API 仕様 を参照のこと。
  "java.text.DecimalFormat"、"java.text.SimpleDateFormat"

- ファイルの 1 行あたりのカラム数についての補足
  ファイル入力の際、アノテーションを設定したカラム数とファイルのカラム数が異なる場合、フレームワークは例外を発生させる。また、固定長ファイル入力の際、アノテーションの bytes で設定したカラムのバイト数の合計と読み取った 1 行のバイト数が異なる場合、フレームワークは例外を発生させる。

- 囲み文字についての補足
  囲み文字が出力項目に含まれている場合、同じ囲み文字を追加してエスケープ編集を行う。また、ファイル入力の際、カラムに囲み文字と同一の文字が含まれておりエスケープ編集されていないカラムの場合、フレームワークは例外を発生させる。

- ファイル上書きフラグについての補足
  FileUpdateDAO を DI したビジネスロジック分割ジョブで起動した場合、ファイル上書きフラグを True に設定するとデータが破損する可能性があることに留意すること。また、分割ジョブにてファイル上書きファイルフラグを false にした場合、複数スレッドから一つのファイルにアクセスするため、データの出力順番がランダムになることに留意すること。

- ファイル入力用 DAO
  フレームワークではファイル入力用 DAO インタフェース、およびファイル入力用イテレータインタフェースを規定し、ファイル形式に対応したそれぞれのデフォルト実装を提供する。
  ファイル入力用 DAO の execute()メソッドを実行し、ファイル入力用イテレータを取得する。ファイルの各行は、ファイル入力用イテレータの next()メソッドで取得する。
  - ファイル入力用インタフェース

項番	インタフェース名	概要
1	jp.terasoluna.fw.file.dao.FileQueryDAO	ファイル入力用 DAO インタフェース
2	jp.terasoluna.fw.file.dao.FileLineIterator	ファイル入力用イテレータインタフェース

  - ファイル入力用 DAO 実装クラス

項番	クラス名	概要
1	jp.terasoluna.fw.file.dao.standard.CSVFileQueryDAO	CSV 形式のファイル入力を行う場合に利用する
2	jp.terasoluna.fw.file.dao.standard.FixedFileQueryDAO	固定長形式のファイル入力を行う場合に利用する
3	jp.terasoluna.fw.file.dao.standard.VariableFileQueryDAO	可変長形式のファイル入力を行う場合に利用する
4	jp.terasoluna.fw.file.dao.standard.PlainFileQueryDAO	文字列データをファイルから入力する場合に利用する

    - ファイル入力用 DAO は、ファイル入力用イテレータを生成する。

  - ファイル入力用イテレータ実装クラス

項番	クラス名	概要
1	jp.terasoluna.fw.file.dao.standard.CSVFileLineIterator	CSV 形式のファイル入力を行う場合に利用する
2	jp.terasoluna.fw.file.dao.standard.FixedFileLineIterator	固定長形式のファイル入力を行う場合に利用する
3	jp.terasoluna.fw.file.dao.standard.VariableFileLineIterator	可変長形式のファイル入力を行う場合に利用する
4	jp.terasoluna.fw.file.dao.standard.PlainFileLineIterator	文字列データをファイルから入力する場合に利用する

    - ファイル入力でのデータ部の入力は、データ部の 1 行分のデータを入出力オブジェクトに格納し、呼び出し元に返却する処理を提供する。
    - ヘッダ部、トレイラ部からの入力用メソッドを提供する。
    - 囲み・区切り文字として設定された文字が入力データの文字例にあると正しく動作しない。

- ファイル出力用 DAO
  フレームワークではファイル出力用 DAO、およびファイル出力用行ライタのインタフェースを規定し、ファイル形式に対応したそれぞれのデフォルト実装を提供する。
  ファイル出力用 DAO の execute()メソッドを実行し、ファイル出力用行ライタを取得する。ファイルの各行は、ファイル出力用行ライタの printDataLine()メソッドで出力する。

  - ファイル出力用インタフェース

項番	インタフェース名	概要
1	jp.terasoluna.fw.file.dao.FileUpdateDAO	ファイル出力用 DAO インタフェース
2	jp.terasoluna.fw.file.dao.FileLineWriter	ファイル出力用行ライタインタフェース

  - ファイル出力用 DAO 実装クラス

項番	クラス名	概要
1	jp.terasoluna.fw.file.dao.standard.CSVFileUpdateDAO	CSV 形式のファイル出力を行う場合に利用する
2	jp.terasoluna.fw.file.dao.standard.FixedFileUpdateDAO	固定長形式のファイル出力を行う場合に利用する
3	jp.terasoluna.fw.file.dao.standard.VariableFileUpdateDAO	可変長形式のファイル出力を行う場合に利用する
4	jp.terasoluna.fw.file.dao.standard.PlainFileUpdateDAO	文字列データをファイルへ出力する場合に利用する

    - ファイル出力用 DAO は、ファイル出力用イテレータを生成する。

  - ファイル出力用行ライタ実装クラス

項番	クラス名	概要
1	jp.terasoluna.fw.file.dao.standard.CSVFileLineWriter	CSV 形式のファイル出力を行う場合に利用する
2	jp.terasoluna.fw.file.dao.standard.FixedFileLineWriter	固定長形式のファイル出力を行う場合に利用する
3	jp.terasoluna.fw.file.dao.standard.VariableFileLineWriter	可変長形式のファイル出力を行う場合に利用する
4	jp.terasoluna.fw.file.dao.standard.PlainFileLineWriter	文字列データをファイルへ出力する場合に利用する

    - ファイル出力でのデータ部の出力は、ファイル行オブジェクトに格納された一行分のデータをファイルに書込む処理を提供する。
    - ヘッダ部、トレイラ部への出力メソッドを提供する
    - ファイル生成時、フォルダ名は存在するフォルダを設定する必要がある。 存在しないとファイルは生成されない。

- ファイル入力チェックについて
  Collector での対象データ取得時に、入力ファイルに対して入力チェックを行うことができる。
  入力チェックについての詳細は、『BD-02 対象データ取得機能』を参照のこと。

- 例外処理
  ファイルアクセス時に例外が発生した場合、ファイルアクセス用の DAO からスローする例外クラスに、エラーが発生したファイルの情報を格納する。例外が発生した処理に対する後処理(処理過程で生成されたファイルの削除処理など) は例外ハンドラで実装すること。例外ハンドラの詳細については『BH-01 例外ハンドリング機能』を参照のこと。
  ファイルアクセス時の例外クラスには、以下の２つのクラスがある。

項番	例外クラス名	概要
1	jp.terasoluna.fw.file.dao. FileException	ファイル全体に関わるエラーに対応する例外クラス。 以下の情報を保持する。 ・ファイル名
2	jp.terasoluna.fw.file.dao. FileLineException （FileException のサブクラス）	ファイルの行に関わるエラーに対応する例外クラス。 以下の情報を保持する。 ・ファイル名 ・エラーが発生した箇所の行番号 ・エラーが発生したカラムのカラムインデックス（0 から開始） ・エラーが発生したカラムのカラム名（ファイル行オブジェクトのプロパティ名）

  例外クラスが保持する情報を用いて、ログ出力などを行うことができる。

■ 使用方法

◆ コーディングポイント

- ファイル行オブジェクトの実装例
  - CSV 形式のデータをファイル行オブジェクトに格納する場合の記述例 (getter/setter は省略)

```
@FileFormat
public class SampleFileLineObject {
     ……
    @InputFileColumn(
           columnIndex = 0,
           columnformat=”yyyy/MM/dd”)
    private Date hiduke = null;

    @InputFileColumn(
           columnIndex = 1,
           stringConverter = StringConverterToUpperCase.class)
    private String shopId = null;

    @InputFileColumn(
           columnIndex = 2,
           columnformat=”###,###,###”)
    private BigDecimal uriage = null;
     ……
}
```

アノテーションの FileFormat は必須

アノテーション InputFileColumn とパラメータの設定

✧ 上記のファイル行オブジェクトに下記の CSV 形式のデータを格納すると、各属性の値は以下の通りとなる。

```
“2006/07/01”,”shop01”,”1,000,000”  ←  CSV形式のデータ
```

✧ ファイル行オブジェクトに設定される値

```
 hiduke = Fri Jul 07 00:00:00 JST 2006
shopId = SHOP01
uriage = 1000000
```

PlainFileQueryDAO、PlainFileUpdateDAO を利用する場合は、@FileFormat のみを記述したファイル行オブジェクトを使用すること。

➢ ファイル全体に関わる定義情報を設定する場合の記述例 (getter/setter は省略)

```
@FileFormat(lineFeedChar="¥r¥n", headerLineCount = 1, trailerLineCount= 1)
public class SampleFileLineObject {
     ……
    @InputFileColumn(
           columnIndex = 0,
           columnformat="yyyy/MM/dd")
    private Date hiduke = null;

    @InputFileColumn(
           columnIndex = 1,
           stringConverter = StringConverterToUpperCase.class)
    private String shopId = null;

    @InputFileColumn(
           columnIndex = 2,
           columnformat="###,###,###")
    private BigDecimal uriage = null;
     ……
}
```

FileFormat で、ヘッダ部行数とトレイラ部行数を指定する。

✧ 上記のファイル行オブジェクトに下記のデータを格納すると、各属性の値は以下の通りとなる。

```
支店名：千葉支店                                ← ヘッダ部
"2006/07/01","shop01","1,000,000"   ← データ部
合計金額：1,000,000                             ← トレイラ部
```

ヘッダ部とトレイラ部を含んだファイル

✧ ファイル行オブジェクトに設定される値

```
hiduke = Fri Jul 07 00:00:00 JST 2006
shopId = SHOP01
uriage = 1000000
```

※ ヘッダ部とトレイラ部はファイル行オブジェクトに格納されない。

➢ ファイル項目でトリム種別を設定し、デフォルトのトリム文字を使用した定義情報を設定する場合の記述例 (getter/setter は省略)

```
@FileFormat()
public class SampleFileLineObject {
     ……
    @InputFileColumn(
           columnIndex = 0,
           columnformat="yyyy/MM/dd")
    private Date hiduke = null;

    @InputFileColumn(
           columnIndex = 1,
           trimType = TrimType.RIGHT,
           stringConverter = StringConverterToUpperCase.class)
    private String shopId = null;

    @InputFileColumn(
           columnIndex = 2,
           columnformat="###,###,###")
    private BigDecimal uriage = null;
     ……
}
```

右側の空白をトリム(削除)するように設定する。

✧ 上記のファイル行オブジェクトに下記の CSV 形式のデータを格納すると、各属性の値は以下の通りとなる。

```
"2006/07/01","shop01   ","1,000,000"  ←  データ部
```

✧ ファイル行オブジェクトに設定される値

```
hiduke = Fri Jul 07 00:00:00 JST 2006
shopId = SHOP01      ←右側にあった空白文字を削除している。
uriage = 1000000
```

➢ ファイル項目でトリム種別を設定し、個別のトリム文字を使用した定義情報を設定する場合の記述例 (getter/setter は省略)

```
@FileFormat()
public class SampleFileLineObject {
     ……
    @InputFileColumn(
           columnIndex = 0,
           columnformat=”yyyy/MM/dd”)
    private Date hiduke = null;

    @InputFileColumn(
           columnIndex = 1,
           trimType = TrimType.LEFT,
           trimChar = ‘0’,
           stringConverter = StringConverterToUpperCase.class)
    private String shopId = null;

    @InputFileColumn(
           columnIndex = 2,
           columnformat=”###,###,###”)
    private BigDecimal uriage = null;
     ……
}
```

左側の’0’の文字ををトリム(削除)するように設定する。

✧ 上記のファイル行オブジェクトに下記の CSV 形式のデータを格納すると、各属性の値は以下の通りとなる。

```
“2006/07/01”,”000shop01”,”1,000,000”  ←  データ部
```

✧ ファイル行オブジェクトに設定される値

```
hiduke = Fri Jul 07 00:00:00 JST 2006
shopId = SHOP01     ←対象文字列の左側にある’0’が削除される。
uriage = 1000000
```

● ジョブ Bean 定義ファイルの設定例

```
<bean id="CSVFile01"
   class="jp.terasoluna.XXX.・・・">
      <!-- 入力ファイルの設定 -->
      <property name="fileDao">
         <ref bean=”csvFileQueryDAO” />
      </property>
```

ファイル入出力用 DAO を使うクラス

ファイル入出力用 DAO 実装クラス。
参照する Bean は「FileAccessBean.xml」を参照のこと

- ファイル入力の実装例
  - ファイル入力処理の実装

    (1) ファイル行オブジェクトを実装する。

    (2) ファイル入力処理を行うクラスのプロパティに、FileQueryDAO 実装クラスを設定する。

    【ジョブ Bean 定義ファイルの設定例】

```
<bean id="blogic"
  class="jp.terasoluna.batch.sample.SampleLogic">
   <property name="fileQueryDAO" ref="csvFileQueryDao" />
</bean>
```

参照する Bean は「FileAccessBean.xml」を参照のこと

    (3) ファイル入力処理を行うクラスでは、FileQueryDAO の execute()メソッドでファイル入力用イテレータを取得する。ファイル入力用イテレータ取得時に、ファイルオープンが行われる。
    ファイル入力用イテレータの next メソッドで、ファイル行オブジェクトを取得する。

    【実装例】

ファイルパスとファイル行オブジェクトクラスを引数にして、ファイル入力用イテレータを取得する

アノテーション FileFormt の headerLineCount で設定した行数分のヘッダ部を取得する

ファイル形式に関わらず、next()メソッドを使用する

アノテーション FileFormt の trailerLineCount で設定した行数分のトレイラ部を取得する

closeFile()メソッドでファイルを閉じること

```
...
   // ファイル入力用イテレータの取得
   FileLineIterator<SampleFileLineObject> fileLineIterator
                     = fileQueryDAO.execute(basePath +
                             "/some_file_path/uriage.csv",FileColumnSa
                                     mple.class);

   try {

     // ヘッダ部の読み込み
     List<String> headerData = fileLineIterator.getHeader();
     ... // 読み込んだヘッダ部に対する処理

     while(fileLineIterator.hasNext()){
        // データ部の読み込み
        SampleFileLineObject sampleFileLine
                                  = fileLineIterator.next();
     ... // 読み込んだ行に対する処理
     }
     // トレイラ部の読み込み
     List<String> trailerData = fileLineIterator.getTrailer();
     ... // 読み込んだトレイラ部に対する処理
   } finally {
      // ファイルのクローズ
      fileLineIterator.closeFile();
   }
...
```

  - ファイルの入力順序

    トレイラ部の入力は、データ部の入力が全て終わった後に行う必要がある点に留意すること。

- スキップ処理
  ファイル入力機能では入力を開始する行を指定できる。これは主に『BE-04 リスタート機能』で中断したジョブを再開する際、リスタートポイントからファイルの読み込みを再開するために利用する。

【ビジネスロジックの実装例】

```
// スキップ処理
……
    fileLineIterator.skip(1000);
……
```

fileLineIterator のカレント行から 1000 行分のデータ行を読み飛ばす処理を行う

- ファイル入力処理の実装（ファイル入力を行うクラスでファイルをオープンしたまま、読み込みを行う場合）
  ファイル入力処理を行うクラスに、(File 更新用 Dao ではなく）ファイル入力用行イテレータを直接設定する。ファイル入力用行イテレータのコンストラクタで、ファイルのパス、ファイル行オブジェクトのクラス等を指定する。

【ジョブ Bean 定義ファイルの設定例】

```
<bean id="blogic" class="testBlogic">
  <property name="iterator">
    <bean class="jp.terasoluna.fw.file.dao.standard.
      CSVFileLineIterator"
      destroy-method="closeFile">
        <constructor-arg index="0" value="some_file_path/uriage.csv" />
        <constructor-arg index="1"
                value="jp.terasoluna.batch.sample.FileColumnSample" />
              <constructor-arg index="2" ref="columnParserMap" />
    </bean>
  </property>
</bean>
```

bean のコンストラクタにファイル名(1 番目の引数。文字型) 、パラメータクラス(2 番目の引数、クラス型)、テキスト変換処理(3 番目の引数。FileAccessBean.xml で定義されている Bean "columnParserMap"を固定で指定) を設定する。

- ファイル入力処理の実装についての補足
  ファイル入力処理を行う場合は、FileQueryDAO を利用してもファイル入力用行イテレータを直接設定する利用する方法のどちらかを利用することで実装出来る。FieQueryDAO を利用する場合は DAO が呼ばれるたびにファイルのオープン/クローズ処理が行われる。そのため、ファイル入力用行イテレータを直接設定した場合と比べて処理が遅くなる。また、ファイル入力用行イテレータを直接設定した場合に、ビジネスロジックで入力処理を行い後処理でファイルの移動などを行うと後処理ではまだファイルストリームが存在するため、エラーが発生することに留意すること。

- ファイル出力の実装例
  - ファイルの設定として、囲み文字と区切り文字を設定し、データの一部をデフォルトのパディング文字でパディング処理したデータをファイルに出力する場合の記述例 (getter/setter は省略)

```
@FileFormat(delimiter = “,”, encloseChar = “¥”” )
public class SampleFileLineObject {
     ……
    @OutputFileColumn(
           columnIndex = 0,
           columnformat=”yyyy/MM/dd”)
    private Date hiduke = null;

    @OutputFileColumn(
           columnIndex = 1,
           paddingType = PaddingType.LEFT,
           bytes = 10,
           stringConverter = StringConverterToUpperCase.class)
    private String shopId = null;

    @OutputFileColumn(
           columnIndex = 2,
           columnformat=”###,###,###”)
    private BigDecimal uriage = null;
     ……
}
```

アノテーションの FileFormat は必須
区切り文字、囲み文字を設定。

アノテーション OutputFileColumn
とパラメータの設定

パディング処理を行う場合は、バイト数の設定が必須となる。

    - ✧ 出力対象となるファイル行オブジェクトの値

```
hiduke = Fri Jul 07 00:00:00 JST 2006
shopId = SHOP01
uriage = 1000000
```

    - ✧ 上記のファイル行オブジェクトを出力すると以下の値となる。

```
“2006/07/01”,”    shop01”,”1,000,000”
```

- データの一部を個別のパディング文字でパディング処理したデータをファイルに出力する場合の記述例 (getter/setter は省略)

```
@FileFormat(delimiter = “,”, encloseChar = “¥”” )
public class SampleFileLineObject {
     ……
    @OutputFileColumn(
           columnIndex = 0,
           columnformat=”yyyy/MM/dd”)
    private Date hiduke = null;

    @OutputFileColumn(
           columnIndex = 1,
           paddingType = PaddingType.RIGHT,
           paddingChar = '0',
           bytes = 10,
           stringConverter = StringConverterToUpperCase.class)
    private String shopId = null;

    @OutputFileColumn(
           columnIndex = 2,
           columnformat=”###,###,###”)
    private BigDecimal uriage = null;
     ……
}
```

右側のパディング処理を行い、パディング文字として'0'を設定する。

✧ 出力対象となるファイル行オブジェクトの値

```
 hiduke = Fri Jul 07 00:00:00 JST 2006
shopId = SHOP01
uriage = 1000000
```

✧ 上記のファイル行オブジェクトを出力すると以下の値となる。

```
“2006/07/01”,”shop010000”,”1,000,000”
```

- ファイル出力処理の実装（1メソッドでファイルをオープン・クローズする場合）

(1) ファイル行オブジェクトを実装する。

(2) ファイル出力処理を行うクラスのプロパティに、FileUpdateDAO 実装クラスを設定する。

【ジョブ Bean 定義ファイルの設定例】

```
<bean id="blogic"
  bean class="jp.terasoluna.batch.sample.SampleLogic">
   <property name="fileUpdateDAO" ref="csvFileUpdateDao" />
</bean>
```

(3) ファイル出力処理を行うクラスでは、FileUpdateDAO の execute メソッドでファイル出力用行ライタを取得する。ファイル出力用行ライタの取得時に、ファイルがオープンされる。

【実装例】

```
...
   // ファイル出力用行ライタの取得
   FileLineWriter< SampleFileLineObject > fileLineWriter
   = fileUpdateDAO.execute(basePath + "/some_file_path/uriage.csv",
                                   SampleFileLineObject.class);
...

try {
   // ヘッダ部の出力
            fileLineWriter.printHeaderLine(headerString);
   ...

   while ( ... ) {
      ...
      // データ部の出力（1行）
      fileLineWriter.printDataLine(sampleFileLineObject);

   }

   // トレイラ部の出力
            fileLineWriter.printTrailerLine(trailerString);
                                   ...
} finally {
   // ファイルのクローズ
   fileLineWriter.closeFile();
}
...
```

ファイル名とパラメータクラスを引数に、ファイル出力用行ライタを取得する。

ヘッダ部を出力する。
String 型の変数を引数とする。

ファイル形式に関わらず、printDataLine メソッドで出力する。
出力される項目には、項目定義用のアノテーションを付加しておく。

トレイラ部を出力する。
String 型の変数を引数とする。

出力が終了したら、ファイルをクローズする。

- ファイル出力処理の実装（ファイル出力を行うクラスでファイルをオープンしたまま、追記する場合）

【ジョブ Bean 定義ファイルの設定例】

```
<bean id="blogic" class="testBlogic">
  <property name="writer">
    <bean class="jp.terasoluna.fw.file.dao.standard.
      CSVFileLineWriter"
      destroy-method="closeFile">
        <constructor-arg index="0" value="some_file_path/uriage.csv" />
        <constructor-arg index="1"
             value="jp.terasoluna.batch.sample.SampleFileLineObject" />
             <constructor-arg index="2" ref="columnFormatterMap" />
    </bean>
  </property>
</bean>
```

bean のコンストラクタにファイル名(1 番目の引数。文字型) 、パラメータクラス(2 番目の引数、クラス型)、テキスト変換処理(3 番目の引数。FileAccessBean.xml で定義されている Bean "columnFormatterMap"を固定で指定) を設定する。

- ファイル出力処理の実装についての補足
  ファイル出力処理を行う場合は、FileUpdateDAO を利用してもファイル出力用行ライタを直接設定する利用する方法のどちらかを利用することで実装出来る。FieUpdateDAO を利用する場合は DAO が呼ばれるたびにファイルのオープン/クローズ処理が行われる。そのため、ファイル出力用行ライタを直接設定した場合と比べて処理が遅くなる。また、ファイル出力用行ライタを直接設定した場合に、ビジネスロジックで出力処理を行い後処理でファイルの移動などを行うと後処理ではまだファイルストリームが存在するため、エラーが発生することに留意すること。

- ファイルの出力順序についての補足
  ヘッダ部の出力は、データ部の出力の前に行う必要がある点に留意すること。
  同様にトレイラ部の出力は、データ部の出力がすべて終わった後に行う必要がある点に留意すること。但し、FileUpdateDAO を前処理、主処理、後処理で利用する場合は順序性が担保出来ているか判断することが出来ないため「トレイラ(前処理)⇒データ(主処理)⇒ヘッダ(後処理)」という出力が可能となる。前処理、主処理、後処理の各々で FileUpdateDAO を利用する場合はヘッダ部、データ部、トレイラ部の出力処理順序の注意が必要である。

- データ部出力についての補足
  ファイルの項目に対する定義情報がないファイル行オブジェクトを printDataLine()メソッドの引数に指定した場合は、ファイルに空文字が書き込まれる。また、行区切り文字も書き込むため printDataLine()メソッドの実行回数分だけ改行コードが書き込まれる。

◆ 拡張ポイント

- なし

■ 関連機能

- 『BD-02 対象データ取得機能』
- 『BE-04 リスタート機能』
- 『BH-01 例外ハンドリング機能』

■ 使用例

- 機能網羅サンプル(functionsample-batch)
- チュートリアル(tutorial-batch)

■ 備考

- なし

BC-02 ファイル操作機能

■ 概要

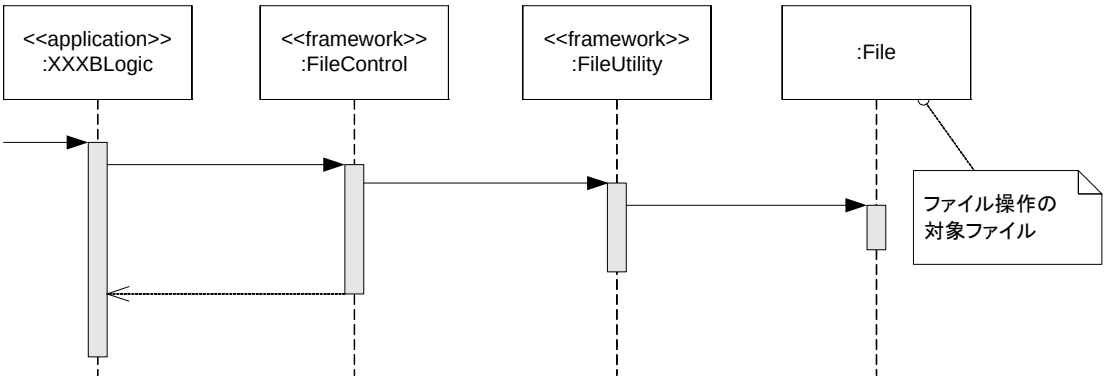
◆ 機能概要

- ファイル操作機能は以下の機能を提供する。
  - ファイル名変更
    変更元ファイルと変更先のファイルを指定して、ファイル名の変更を行う。
  - ファイル移動
    ファイルの移動は「ファイル名変更」を利用し、パスを変更することにより実現する。
  - ファイルコピー
    コピー元ファイルとコピー先ファイルを指定して、ファイルのコピーを行う。
  - ファイル削除
    削除するファイルを指定して、ファイルの削除を行う。
  - ファイル結合
    指定されたファイル名のリストにあるファイルを結合する。

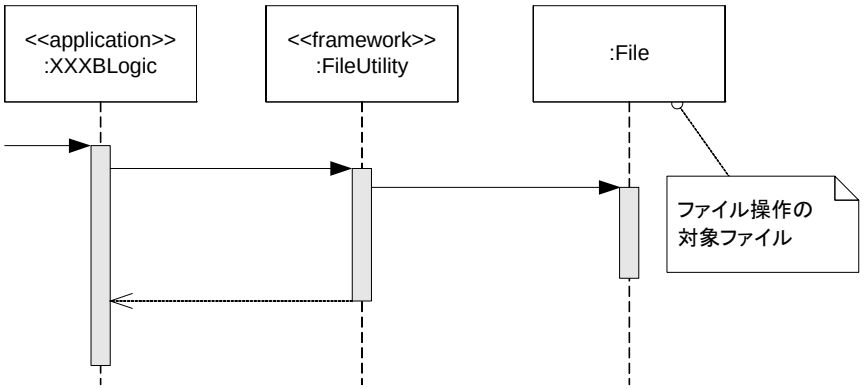
◆ 概念図

ファイル操作の概念図。

- ビジネスロジックからファイル操作インタフェースを利用して、ファイル操作を行なう場合



- ビジネスロジックからファイル操作ユーティリティクラスを直接利用して、ファイル操作を行なう場合



◆ 解説

- FileControl
  ファイル操作の処理を提供するインタフェース。インタフェースが提供するメソッドの詳細についてはフレームワークの JavaDoc を参照のこと。ここでは、FileControl インタフェースで提供するファイル操作の代表的なメソッドを下記に挙げる。
  - ファイル操作インタフェース

項番	インタフェース名	概要
1	jp.terasoluna.fw.file.util.FileControl	ファイル操作用のインタフェース

  - ファイル操作インタフェースで提供する代表的なメソッド

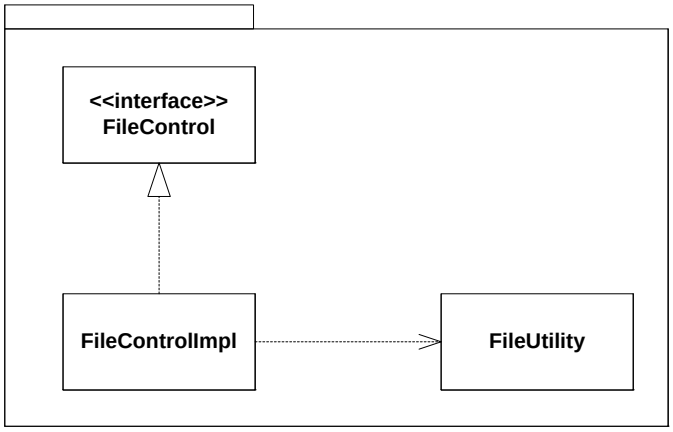
項番	メソッド	概要
1	renameFile(String scrFile, String newFile)	ファイル名の変更・ファイルの移動
2	copyFile(String scrFile, String newFile)	ファイルのコピー
3	deleteFile(String scrFile)	ファイルの削除
4	mergeFile(List<String> fileList, String newFile)	fileList にあるファイルの結合

- FileControlImpl
  FileControl インタフェースを実装するクラス。FileControlImpl クラスはビジネスロジックに DI して利用することができる。FileControlImpl クラスはファイル操作処理を実行する FileUtility クラスをラップしている。
  FileControlImpl クラスは、属性にファイル操作を行う際に基準となるパス（基準パス）を持つ。基準パスを使用することにより、ファイルアクセス時に発生するファイルパスの環境依存の問題を回避することができる。ファイル操作を行うメソッドの引数には相対パス、または、絶対パスを設定する。ファイル操作時にエラーが発生した場合、非検査例外をスローする。
  - 基準パスはファイル操作機能を使う上での基準となる位置を指す。基準パスを「/si1/」、相対パスを「chohyo/test.txt」とした場合、ファイルの絶対パスは「/si1/chohyo/test.txt」となる。

- FileUtility
  ファイル操作機能を実装するクラス。FileUtility のメソッドをビジネスロジックから直接利用することも可能である。FileUtility クラスで提供するメソッドは FileControl インタフェースで提供するものと同じになる。FileUtility クラスではファイル操作を行うメソッドの引数は絶対パスのみ設定可能である。ファイル操作時にエラーが発生した場合、非検査例外をスローする。
  - ファイル操作ユーティリティクラス

項番	メソッド	概要
1	`jp.terasoluna.fw.file.util.FileUtility`	ファイル操作ユーティリティクラス

- ファイル操作機能のクラス図
  - FileControl、FileControlImpl、FileUtilityクラスの関連は下記のクラス図のとおりである。



■ 使用方法

◆ コーディングポイント

- ファイル操作クラス（FileControl）を利用する例
  - Bean定義ファイルの設定例

```
……
<bean id="fileControl"
    class="jp.terasoluna.fw.file.util.FileControlImpl">
    <property name="basePath" value="${basepath}" />
    <property name="checkFileExist" value="false" />
</bean>
……
<bean id="sampleBLogic"
    class="jp.terasoluna.batch.sample.SampleBLogic">
……
    <property name="fileControl" ref="fileControl"/>
……
```

FileControl インタフェースを実装するクラスをフレームワーク Bean 定義ファイルに定義する。プロパティに基準パスを設定すること。

操作後にできるファイルパスにファイルが存在する場合、処理を継続する(上書きする：true)か例外を投げて停止する(false)かを決めるフラグ。

フレームワーク Bean 定義ファイルで設定した Bean を参照する

  - ビジネスロジックの実装例（ファイルのコピー、移動、削除処理の実装例）

```
……
private FileControl fileControl = null;
public void setFileControl(FileControl fileControl){
    this.fileControl = fileControl;
}
……
// ファイルのコピー（相対パスを設定する例）
//  /si1/chohyo/test.txt を/si1/chohyo/testFile.txt にコピー。
// 基準パスは「/si1/」
fileControl.copyFile("chohyo/test.txt", "chohyo/testFile.txt");
……
// ファイルの移動（相対パスを設定する例）
//  /si1/chohyo/testFile.txt を/si1/output/testFile.txt に移動。
// 基準パスは「/si1/」
fileControl.renameFile("chohyo/testFile.txt", "output/testFile.txt");
……
//ファイルの削除（相対パスを設定する例）
//  /si1/chohyo/testFile.txt を削除。
//基準パスは「/si1/」
fileControl.deleteFile("chohyo/testFile.txt");
……
// ファイルのコピー（絶対パスを設定する例）
//  /si1/chohyo/test.txt を/si1/chohyo/testFile.txt にコピー。
fileControl.copyFile("/si1/chohyo/test.txt", "/si1/chohyo/testFile.txt");
……
```

ファイル操作機能を利用するクラスは、FileControl インタフェースとその setter が必須

各メソッドの引数はファイルの相対パス、もしくは絶対パスを記述する

- ビジネスロジックの実装例（ファイル結合の実装例）

```
……
// ファイルの結合。
// 以下に挙げるファイルをリストに格納し、ファイルを/si1/output/mergeFile.csv に統合。
//  /si1/chohyo/output001.csv
//  /si1/chohyo/output002.csv
//  /si1/chohyo/output003.csv
// 基準パスは「/si1/」
fileList.add("chohyo/output001.csv");
fileList.add("chohyo/output002.csv");
fileList.add("chohyo/output003.csv");
……
fileControl.mergeFile(fileList, "output/mergeFile.csv");
……
```

メソッドの 2 番目の引数はファイルの相対パス、もしくは絶対パスを記述する

- ファイル操作ユーティリティクラス（FileUtility）を直接利用する例
  - ビジネスロジックの実装例（ファイルのコピー、移動、削除処理の実装例）

```
……
// ファイルのコピー。
//  /si1/chohyo/test.txt を/si1/chohyo/testFile.txt にコピー。
FileUtility.copyFile("/si1/chohyo/test.txt", "/si1/chohyo/testFile.txt");
……
// ファイルの移動。
//  /si1/chohyo/testFile.txt を/si1/output/testFile.txt に移動。
FileUtility.renameFile("/si1/chohyo/testFile.txt",
    "/si1/output/testFile.txt");
……
//ファイルの削除。  /si1/chohyo/testFile.txt を削除。
FileUtility.deleteFile("/si1/chohyo/testFile.txt");
……
```

各メソッドの引数はファイルの**絶対パス**を記述する

◆ 拡張ポイント

- なし。

■ 関連機能

- なし。

■ 使用例

- 機能網羅サンプル(functionsample-batch)
- チュートリアル(tutorial-batch)

■ 備考

- なし。

BD-01 ビジネスロジック実行機能

■ 概要

◆ 機能概要

- ビジネスロジックの実行方式を提供する。
  - １つのジョブに対して、１つのビジネスロジックが定義できる。
  - ビジネスロジッククラスは、TERASOLUNA-Batch の BLogic インタフェースを実装して作成する。

- ビジネスロジックの入力パラメータには、『BD-02 対象データ取得機能』で作成した POJO が渡される。
  - BLogic インタフェースは入力パラメータをジェネリクス（総称）で定義している。ビジネスロジックを作成する際には、入力パラメータとして受け取る POJO クラスを型パラメータとして定義する。

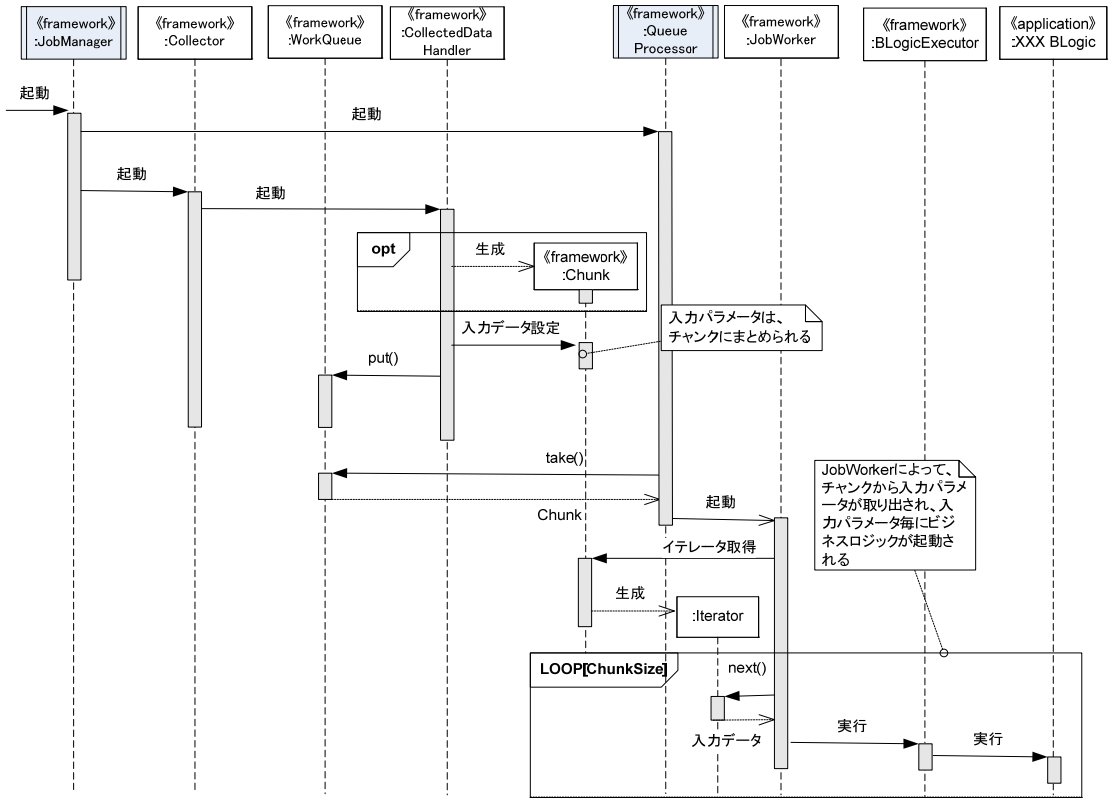
- ビジネスロジックは、実行結果として BLogicResult を返却する。
  - BLogicResult はリターンコードを持ち、ビジネスロジック処理の継続、中止などをフレームワークに指示する。

- ビジネスロジックを実行する際に、後続のビジネスロジックの実行に引き継ぐ情報、あるいは前処理／後処理と共有する情報はジョブコンテキストに設定することができる。
  - ジョブコンテキストは、フレームワークで提供する基底クラスであり、拡張することができる。基底クラスを拡張したジョブコンテキストを使う場合は個別のジョブを定義する際に使用するジョブコンテキストクラスを設定する必要がある。
  - ジョブコンテキストは、ビジネスロジックの入力パラメータとして渡される。

- ジョブに対しては、ビジネスロジックのほかにジョブ前処理／ジョブ後処理、および先頭チャンク前処理／最終チャンク後処理が任意で定義できる。
  - それぞれの前処理／後処理は、それぞれフレームワークで定めたインタフェースを実装して作成する。
  - 先頭チャンク前処理／最終チャンク後処理は、全チャンク単一トランザクションモデルのジョブのみ設定できる。

◆ 概念図

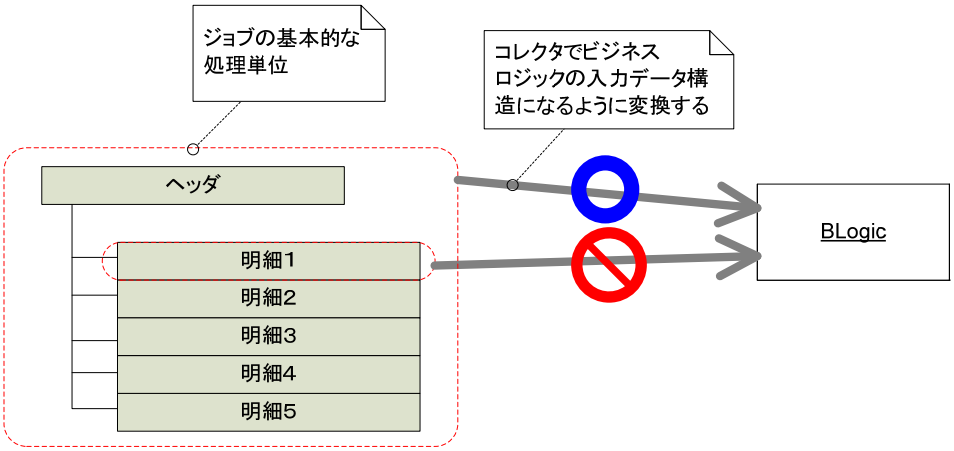
- 対象データ取得から、ビジネスロジック実行までの流れ
  Collector で取得されたデータは、チャンクサイズ毎にチャンクにまとめられる。
  チャンク内の各データ毎に、ビジネスロジックが起動される。



◆ 解説

- 入力パラメータのクラス
  入力パラメータは、Collector によってデータベースやファイルから取得されたデータである。
  ビジネスロジックの入力パラメータクラスは、Collector で取得するデータのクラスと一致させる必要がある。

- 入力パラメータのデータ構造
  ジョブを実行する際には、ビジネスロジックが論理トランザクション単位となる。論理トランザクション単位とは、DBMS 等の物理トランザクション処理の最小単位である。論理トランザクションを複数まとめたものが物理トランザクションとなる。
  ビジネスロジックが論理トランザクションとなるように、必要に応じて Collector でデータ構造を構成する必要がある。
  たとえば、下図のようなヘッダと明細からなるデータ構造において、これらを別のトランザクションでは処理できない場合には、ヘッダと明細をまとめてビジネスロジックの入力となるようにする。ヘッダと明細を一つのトランザクションで処理しなければならないのに、それぞれを別のビジネスロジック呼び出しの入力パラメータとしてしまうと、一部の明細のトランザクションのみがコミットされるなどの不整合が生じる。



データベースから読み込んだ結果を 1:N の構造（ヘッダ-明細の構造）のデータに変換するためには、iBATIS の機能を利用できる。詳細は、iBATIS のドキュメントを参照のこと。

- ビジネスロジックの実行結果の返却
  ビジネスロジックの実行結果は、BLogicResult で返却する。BLogicResult には、あらかじめ定義されている以下のリターンコードを設定する。

項番	リターンコード	バッチ更新	概要
1	NORMAL_CONTINUE	登録可	処理を継続する。通常はこのリターンコードを返却する。 最後の入力パラメータである場合にはトランザクションをコミットし、ジョブを正常終了する。
2	NORMAL_END		対象データの途中で処理を終了させたい場合に返却する。 トランザクションをコミットし、ジョブを正常終了する。 (分割ジョブである場合には、子ジョブのみ終了する)
3	ERROR_CONTINUE	登録不可	エラーデータとしてログ出力を行い、処理を継続する。 ビジネスロジックが含まれるトランザクションには影響を与えない。
4	ERROR_END		エラーメッセージをログに出力し、ジョブを終了する。 トランザクションはロールバックされる。 (分割ジョブである場合には、子ジョブのみ終了する)

ビジネスロジックの実行結果として返却された BLogicResult は、処理結果ハンドラによって処理される。デフォルトの処理結果ハンドラ実装ではエラーデータのログ出力を行う。また、デフォルトのバッチ更新プロセッサにより、バッチ更新登録された SQL の処理を行う。
バッチ更新とは、複数の SQL の処理を一度に処理させることができる機能である。一度に処理させることで、パフォーマンスが大きく向上することがある。
リターンコードにて ERROR_CONTINUE を指定した場合、「ビジネスロジックをまたがった JDBC バッチ更新」を利用して SQL を発行するときと、ビジネスロジック内で SQL を発行したときで挙動が異なるため注意が必要である。
バッチ更新を利用したときは、バッチ更新に登録しようとした SQL は破棄される。一方、ビジネスロジック内で SQL を発行したときは、リターンコードが ERROR_CONTINUE であったとしてもすでに SQL は実行されており、トランザクションのコミット/ロールバックは、後続処理に依存する。
BLogicResult の処理結果ハンドラによる処理については、『BE-05 処理結果ハンドリング機能』を参照のこと。バッチ更新登録については、『BB-01 データベースアクセス機能』を参照のこと。

- ジョブ終了コードの返却
  ビジネスロジックで実行結果を返却する時にはジョブ終了コードを任意で指定することができる。ジョブ終了コードは、Java VM の終了コード（同期型ジョブ起動の場合)、あるいはジョブ管理テーブルに登録されるジョブ終了コード（非同期型ジョブ起動の場合）となる。

項番	リターンコード	ジョブ終了コードの指定	未指定時のジョブ終了コード
1	NORMAL_CONTINUE	無効	-
2	NORMAL_END	有効	デフォルト終了コード（ENDING_NORMARLLY）
3	ERROR_CONTINUE	無効	-
4	ERROR_END	有効	デフォルト終了コード（ENDING_ABNORMARLLY）

  ジョブ終了コードについては、『BE-05 処理結果ハンドリング機能』を参照のこと。同期型ジョブ起動、非同期型ジョブ起動については、それぞれ『BE-01 同期型ジョブ起動機能』『BE-02 非同期型ジョブ起動機能』を参照のこと。

- ジョブコンテキストの利用方法
  ジョブコンテキストは、ビジネスロジック、および前処理／後処理で共有される。
  ジョブコンテキストは、フレームワークから提供する基底クラスを拡張して作成し、ジョブ Bean 定義ファイルで設定する。拡張が必要ではなく基底クラスをそのまま利用する場合はジョブコンテキストのジョブ Bean 定義は不要となる。
  たとえば、前処理で運用日付を取得しジョブコンテキストにセットしておき、ビジネスロジックではジョブコンテキストから運用日付を取得する、というような使い方ができる。その場合には、運用日付のセッタ／ゲッタを持つよう、ジョブコンテキストを拡張する。

- ジョブコンテキストの基底クラス
  ジョブコンテキストは、フレームワークが提供する基底クラスを拡張して作成する。

項番	基底クラス	概要
1	jp.terasoluna.fw.batch.openapi.JobContext	ジョブ ID などのジョブの基本的な情報を、フレームワークからジョブコンテキストに設定するための基底クラス

➢ JobContext 基底クラスによって設定されるデータ項目

通常ジョブ、および分割ジョブの子ジョブのジョブコンテキストには、JobContext 基底クラスを通して、フレームワークから以下のデータ項目が設定される。

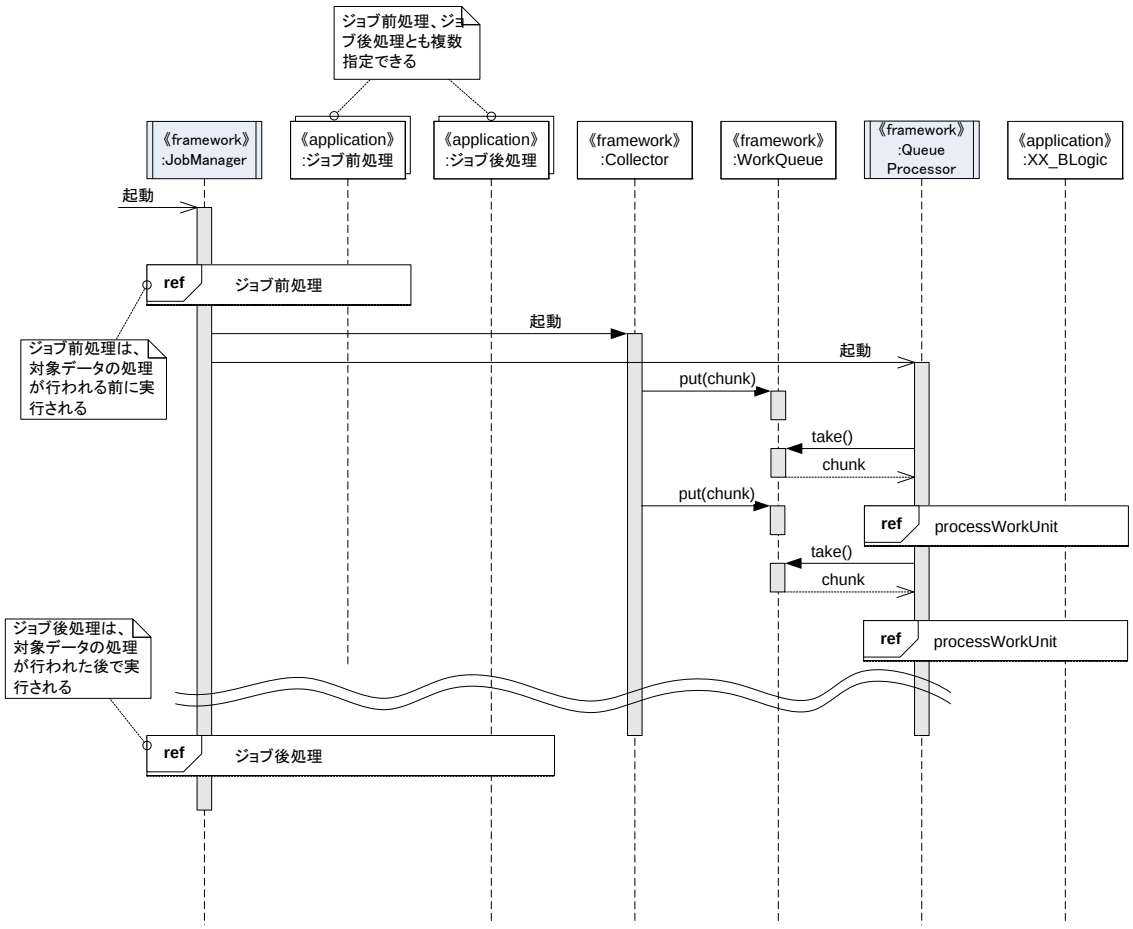
項番	JobContext 基本クラスによって設定されるデータ項目	補足
1	ジョブ ID	システム内でジョブを一意に識別する ID。
2	起動種別 （同期型ジョブ起動、 非同期型ジョブ起動）	ジョブがどのように起動されたかを示す種別。
3	ジョブ依頼番号	非同期型ジョブ起動時は、ジョブ管理テーブルに登録された際のジョブ依頼番号。 同期型ジョブ起動時は、プロセスID（指定されたときのみ）
4	リスタート可能	ジョブがリスタート可能であるかどうかのフラグ。 リスタートが可能である場合には、true。
5	リスタート区分	ジョブがリスタートとして実行されたかどうかのフラグ。 リスタートポイントから再開した実行である場合には、true。
6	リスタートポイント	ジョブを再開するポイント。ジョブがリスタートとして実行された場合にのみ設定される。
7	分割番号	分割ジョブの子ジョブである場合に、その子ジョブが処理する対象データの分割番号（連番）。 （分割ジョブの場合のみ）

ジョブ依頼番号については、『BE-02 非同期型ジョブ起動機能』を参照のこと。
リスタート区分については、『BE-04 リスタート機能』を参照のこと。

➢ 分割ジョブのジョブコンテキスト

分割ジョブでは、親ジョブ、子ジョブは、それぞれ親ジョブ用ジョブコンテキスト、子ジョブのジョブコンテキストを持つ。子ジョブ用のジョブコンテキストは、親ジョブ用ジョブコンテキストから作成される。子ジョブのジョブコンテキストの作成には、JobContext クラスの getChildJobContext()メソッドが使用される。
子ジョブは多重実行されるため、それぞれ別インスタンスの子ジョブ用コンテキストが作成される。

- ジョブ前処理／ジョブ後処理
  ジョブには、ジョブ前処理／ジョブ後処理が任意で設定できる。
  ジョブ前処理は、ジョブが起動された後、かつ Collector による対象データ取得が開始される前に実行される。ジョブ後処理は、対象データの処理が終了した後、かつジョブが終了する前に実行される。
  ジョブ前処理とジョブ後処理は、１つのジョブに対してそれぞれ複数指定できる。複数のジョブ前処理／ジョブ後処理を指定した場合には、順次起動される。必要がなければ、１つも指定しなくともよい。
  ジョブ前処理とジョブ後処理のトランザクションについては、『BA-01 トランザクション管理機能』を参照のこと。



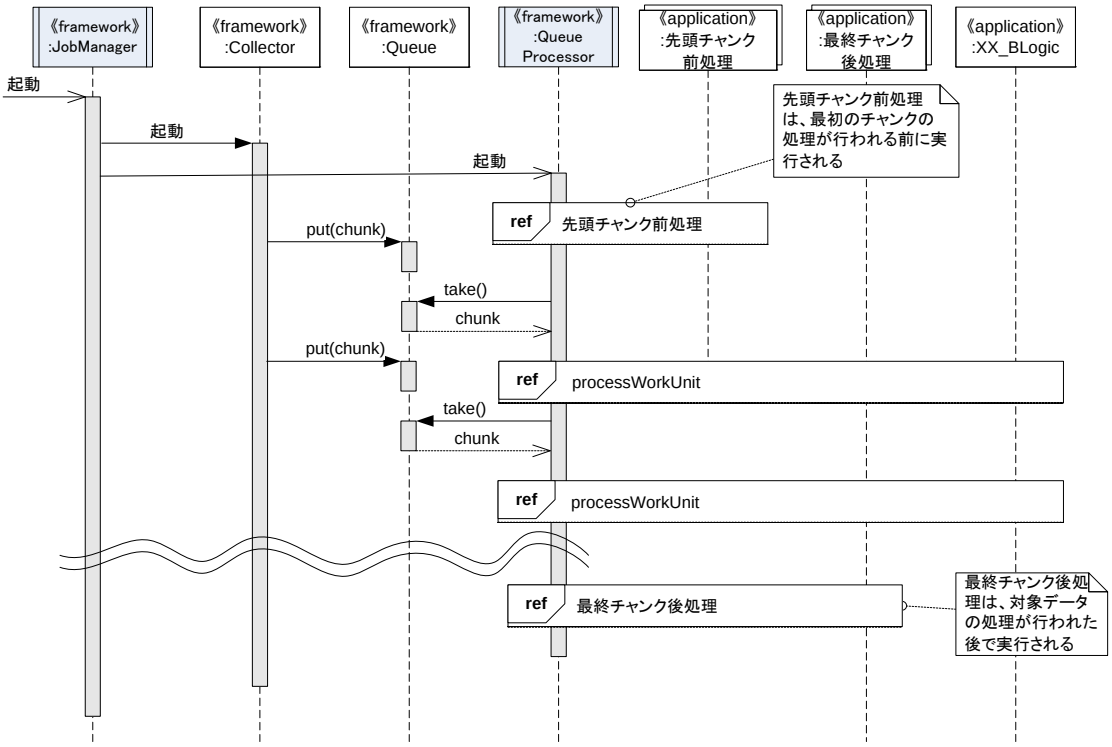
- ジョブ前処理／ジョブ後処理の実行結果の返却
  前処理／後処理においても、ビジネスロジックと同様に BLogicResult を返却する。
  ジョブ前処理／ジョブ後処理から返却された BLogicResult は、ビジネスロジックと同様に処理結果ハンドラによって処理される。

- ジョブ前処理／ジョブ後処理、およびビジネスロジックの起動
  ジョブ前処理／ジョブ後処理、およびビジネスロジックにおいて、NORMAL_END（正常終了）または ERROR_END（異常終了）が返却された場合には、フレームワークによって以下のように処理される。

項番	NORAML_END または ERROR_END を返却した処理	フレームワークによる処理
1	ジョブ前処理が NORMAL_END または ERROR_END を返却した場合	・ジョブ前処理が複数設定されていた場合であっても、後続のジョブ前処理は起動されない。 ・ビジネスロジックは起動されず、ジョブ後処理も起動されない。
2	ビジネスロジックが NORMAL_END または ERROR_END を返却した場合	・未処理の処理対象データが残っていた場合であっても、後続のデータに対するビジネスロジックは起動されない。 ・ジョブ後処理は起動されない。
3	ジョブ後処理が NORMAL_END または ERROR_END を返却した場合	・後続のジョブ後処理は起動されない。

- 分割ジョブでのジョブ前処理／ジョブ後処理
  ジョブ前処理とジョブ後処理は、親ジョブと子ジョブのどちらに対しても指定できる。
  親ジョブでは、ジョブを分割するための分割キー取得前にジョブ前処理が行われ、全ての分割キーの処理が正常・異常に関わらず、終了後（全ての子ジョブが終了後）にジョブ後処理が行われる。ただし、子ジョブの起動に失敗した場合は親ジョブの後処理は実行されない。

- 先頭チャンク前処理、最終チャンク後処理
  ジョブが全チャンク単一トランザクションモデルである場合には、ジョブ前処理／ジョブ後処理とは別に、先頭チャンク前処理、最終チャンク後処理を定義することができる。
  先頭チャンク前処理は、ジョブ前処理が終了後（ジョブ前処理が設定されていた場合)、先頭のチャンクの処理が開始される前に起動される。最終チャンク後処理は、最終チャンクの処理が終了後、ジョブ後処理が起動される前（ジョブ後処理が設定されていた場合）に起動される。



  先頭チャンク前処理、最終チャンク後処理とも、１つのジョブに対して複数指定できる。複数の先頭チャンク前処理、最終チャンク後処理を指定した場合には、順次起動される。必要がなければ、一つも指定しなくともよい。
  先頭チャンク前処理と最終チャンク後処理のトランザクションについては、『BA-01 トランザクション管理機能』を参照のこと。

- 先頭チャンク前処理、最終チャンク後処理の実行結果の返却
  先頭チャンク前処理、最終チャンク後処理においても、ビジネスロジックと同様に BLogicResult を返却する。
  先頭チャンク前処理、最終チャンク後処理から返却された BLogicResult は、ビジネスロジックと同様に処理結果ハンドラによって処理される。

- 先頭チャンク前処理、最終チャンク後処理、およびビジネスロジックの起動
  先頭チャンク前処理、最終チャンク後処理において、NORMAL_END（正常終了） または ERROR_END（異常終了）が返却された場合には、フレームワークによって以下のように処理される。

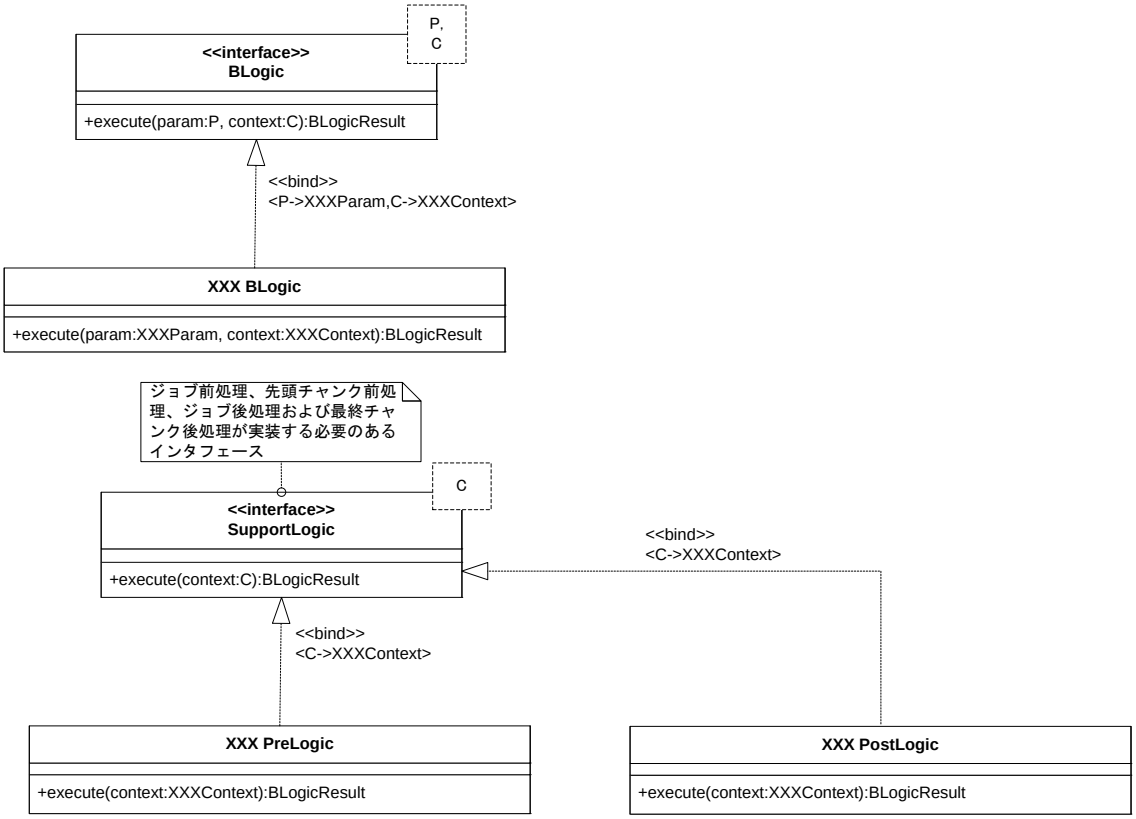
項番	NORMAL_END または ERROR_END を返却した処理	フレームワークによる処理
1	先頭チャンク前処理が NORMAL_END または ERROR_END を返却した場合	・先頭チャンク前処理が複数設定されていた場合であっても、後続の先頭チャンク前処理は起動されない。 ・ビジネスロジックは起動されない。 ・ジョブ後処理は起動されない。
2	ビジネスロジックが NORMAL_END または ERROR_END を返却した場合	・後続のデータに対するビジネスロジックは起動されない。 ・最終チャンク後処理は起動されない。 ・ジョブ後処理は起動されない。
3	最終チャンク後処理が NORMAL_END または ERROR_END を返却した場合	・最終チャンク後処理が複数設定されていた場合であっても、後続の最終チャンク後処理は起動されない。 ・ジョブ後処理は起動されない。

- 分割ジョブでのビジネスロジック実装上の注意（スレッドセーフ性）
  分割ジョブで用いるビジネスロジックは、複数のスレッドで BLogic クラスの一つのインスタンスが実行される。したがって、分割ジョブで用いるビジネスロジックはスレッドセーフで実装する必要がある。

- 共通処理
  共通処理（ビジネスロジック、あるいは前処理/後処理から呼ばれる処理）は、Bean 定義を行ってビジネスロジックに DI することで利用する。

- ビジネスロジックの例外処理
  ビジネスロジック、あるいは前処理／後処理から、フレームワークに例外がスローされてきた場合の処理については、『BH-01 例外ハンドリング機能』を参照のこと。

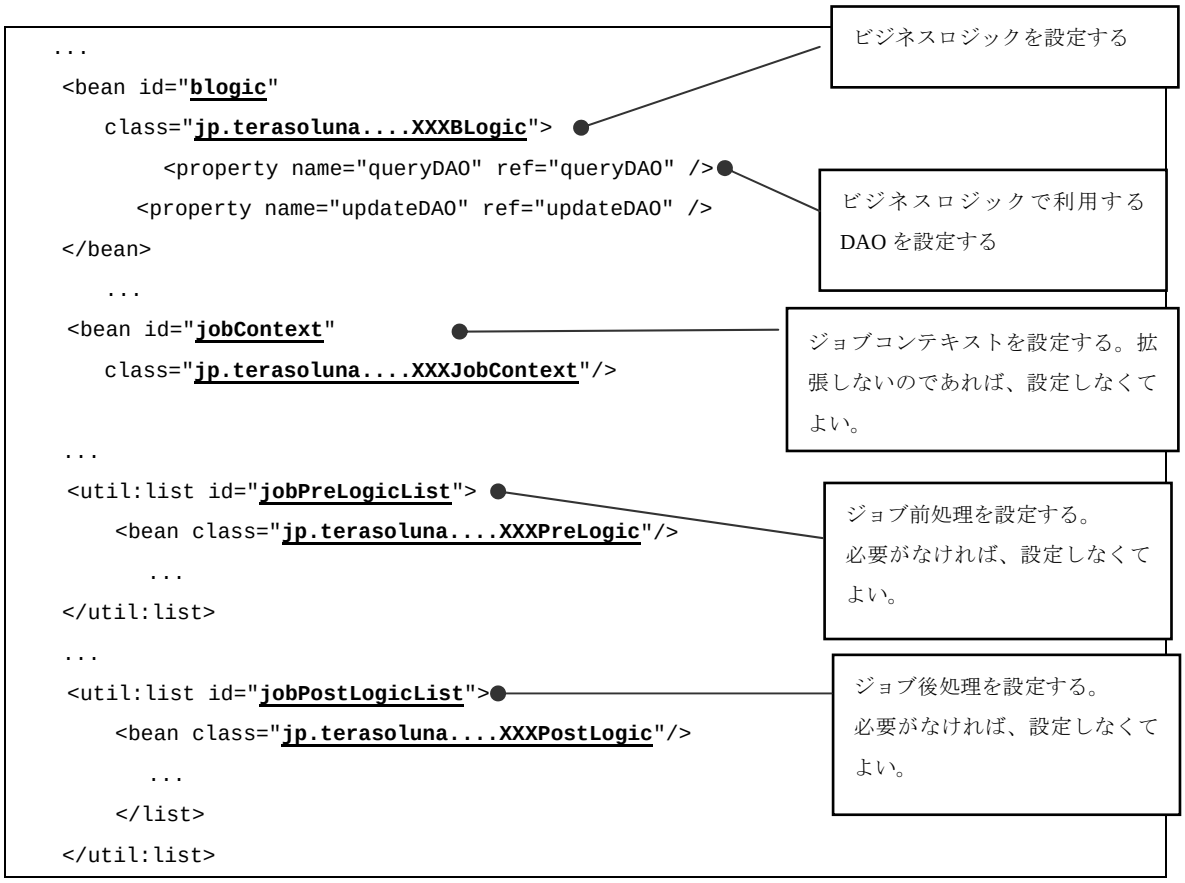
- ビジネスロジック、および前処理／後処理のクラス構造
  ビジネスロジック、および前処理／後処理のクラス構造は、それぞれフレームワークで定めるインタフェースを実装して作成する。



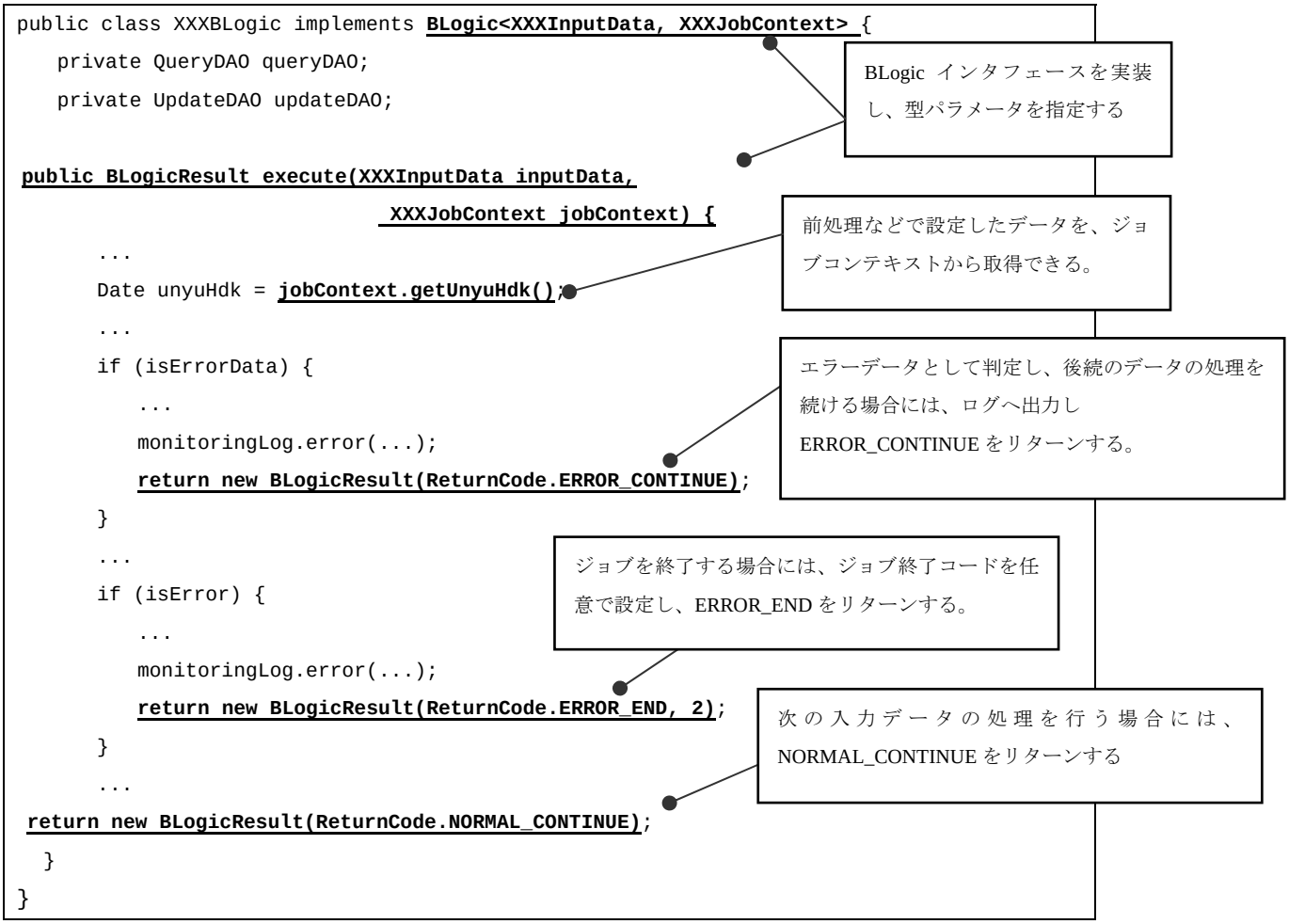
■ 使用方法

◆ コーディングポイント

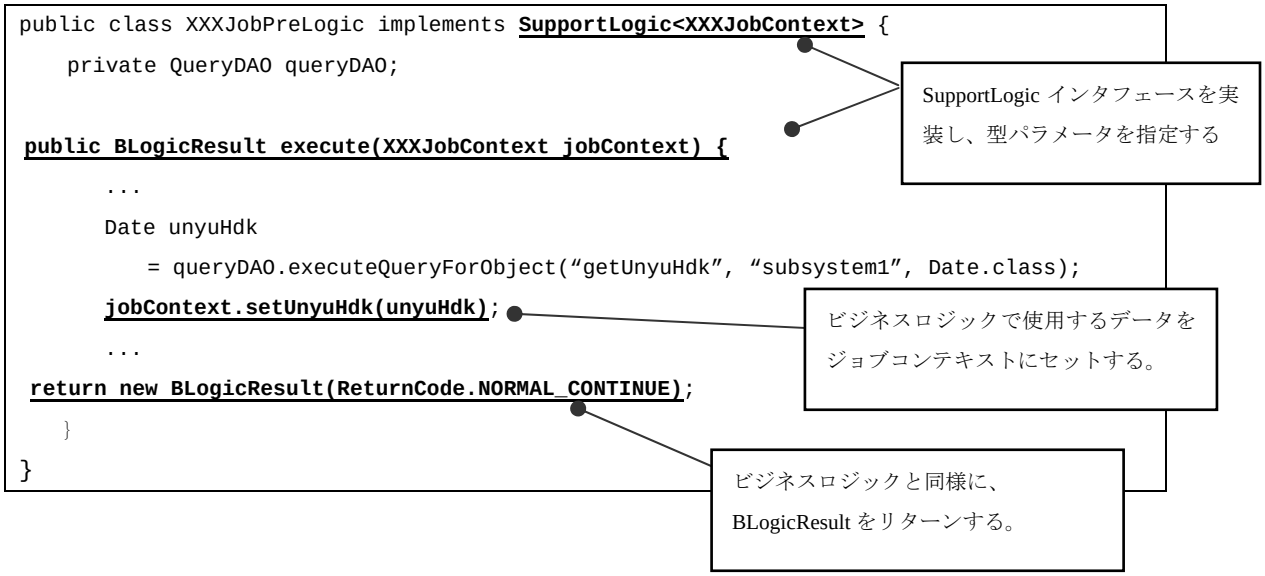
- ジョブ Bean 定義ファイルでのビジネスロジックの設定
  開発者が作成したビジネスロジックをジョブ Bean に設定する。
  - ジョブ Bean 定義ファイル設定例



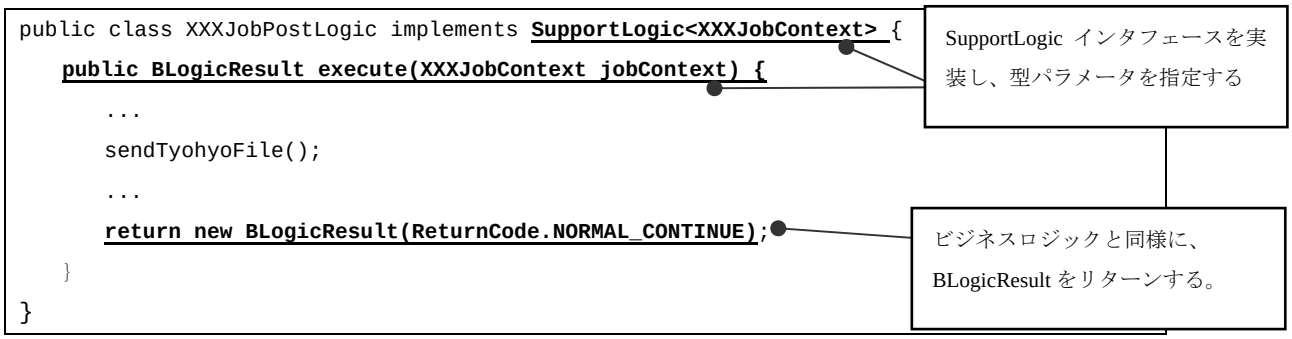
- ビジネスロジックの実装
  TERASOLUNA-Batch が提供する BLogic インタフェースを実装し、ビジネスロジックを作成する。
  - ビジネスロジックの実装例



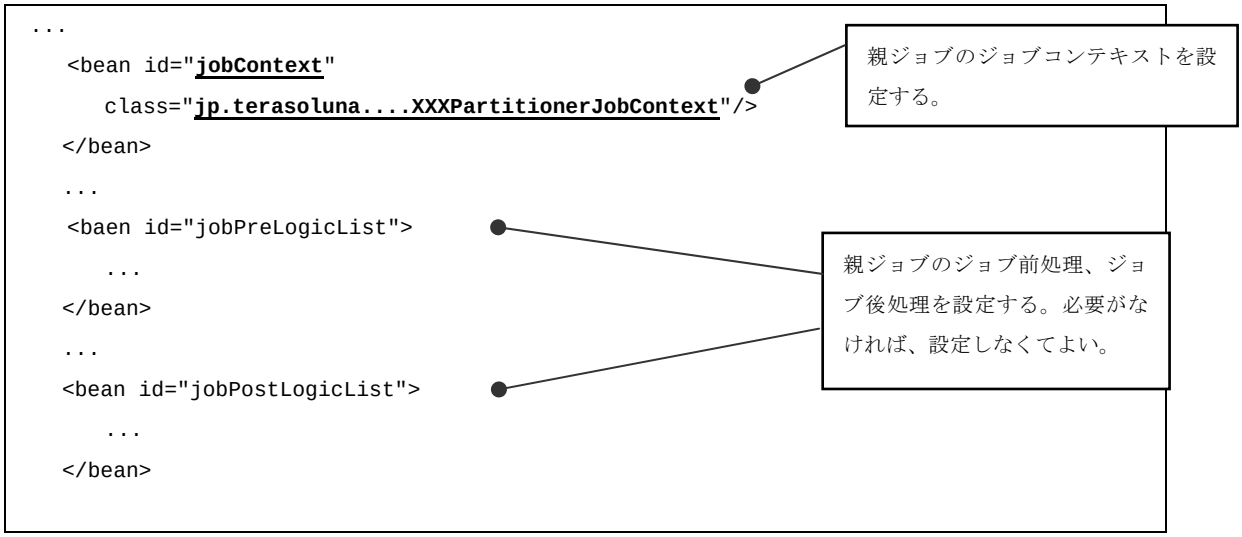
- ジョブ前処理の実装
  フレームワークが提供するインタフェースを実装し、ジョブ前処理を作成する。
  - ジョブ前処理の実装例



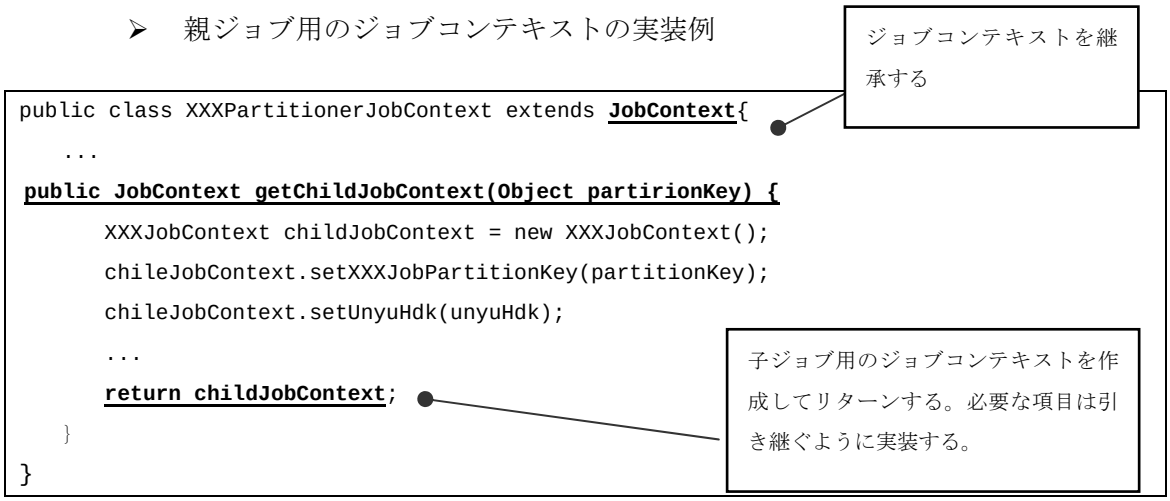
- ジョブ後処理の実装
  フレームワークが提供するインタフェースを実装し、ジョブ後処理を作成する。
  - ジョブ後処理の実装例



- 分割ジョブでのジョブコンテキストの設定
  開発者が作成したビジネスロジックをジョブ定義に設定する。ジョブコンテキストを拡張しない場合は設定する必要がない。
  - ジョブ Bean 定義ファイル設定例



  - 親ジョブ用のジョブコンテキストの実装例



- ジョブ Bean 定義ファイルでの先頭チャンク前処理・最終チャンク後処理の設定
  全チャンク単一トランザクションモデルには、先頭チャンク前処理・最終チャンク後処理をジョブ Bean に設定することができる。
  - ジョブ Bean 定義ファイル設定例

```
<import resource="../../template/singleTransactionBean.xml"/>
...
<bean id ="blogic">
...
</bean>
...
<bean id="jobContext">
...
</bean>
...
 <util:list id="firstchunkPreLogicList">
    <bean class="jp.terasoluna....XXXPreLogic"/>
    ...
</util:list>
...
<util:list id="lastchunkPostLogicList">
    <bean class=" jp.terasoluna....XXXPostLogic" />
    ...
</util:list>
```

全チャンク単一トランザクションモデルである場合のみ、先頭チャンク前処理・最終チャンク後処理を設定することができる

先頭チャンク前処理を設定する。必要がなければ、設定しなくてよい。

最終チャンク後処理を設定する。必要がなければ、設定しなくてよい。

◆ 拡張ポイント

- 前処理、本処理時のエラー発生有無に関わらない後処理の実行
  フレームワークが提供している後処理実行クラス jp.terasoluna.fw.batch.standard.StandardSupportProcessor の process メソッドでは、ジョブ状態を確認し後処理を実行するかどうかを判定している。
  前処理、本処理時のエラー発生有無に関わらず後処理を必ず実行したい場合、後処理実行クラスを拡張することによって実現できる。
  ただし、前処理、対象データ取得処理、本処理などにて例外が発生し、フレームワークが『BH-01 例外ハンドリング機能』によって例外を処理しきれなかった場合は後処理実行クラスを拡張したとしても実行されないので注意すること。

■ 関連機能

- 『BA-01 トランザクション管理機能』
- 『BD-02 対象データ取得機能』
- 『BE-01 同期型ジョブ起動機能』
- 『BE-02 非同期型ジョブ起動機能』
- 『BE-05 処理結果ハンドリング機能』
- 『BF-01 メッセージ管理機能』

■ 使用例

- 機能網羅サンプル(functionsample-batch)
- チュートリアル(tutorial-batch)

■ 備考

- なし。

BD-02 対象データ取得機能

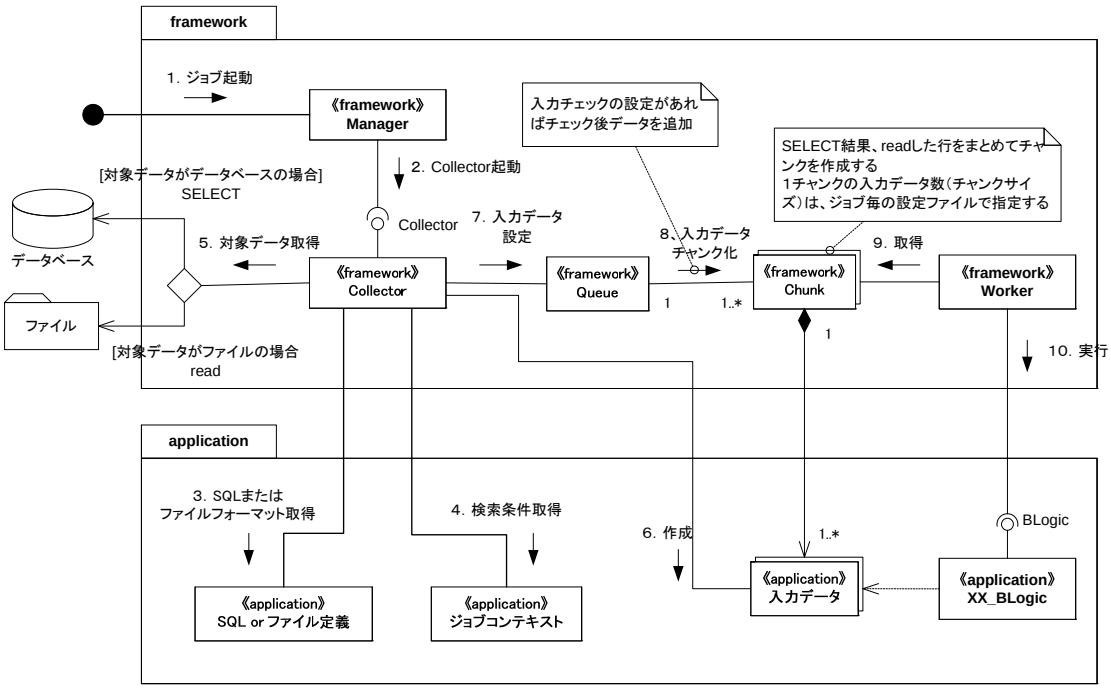
■ 概要

◆ 機能概要

- ジョブの処理対象データの取得機能を提供する。
  - 処理対象データは、ビジネスロジックの入力パラメータとなる。
  - Collector インタフェースを提供し、対象データ取得の詳細を隠蔽する。
  - Commons Validator による入力データチェック機能を提供する。
  - 分割ジョブでは、分割を行うジョブ（親ジョブ）において分割キーを取得する。
- フレームワークでは、Collector インタフェースのデフォルト実装として以下のクラスを提供する。
  - データベースからの対象データ取得機能
  - ファイル（CSV 形式、固定長形式、可変長形式）からの対象データ取得機能
  - ジョブ Bean 定義ファイル（リスト形式、文字列配列形式）からの対象データ取得機能

◆ 概念図

- 対象データ取得の静的構造



◆ 解説

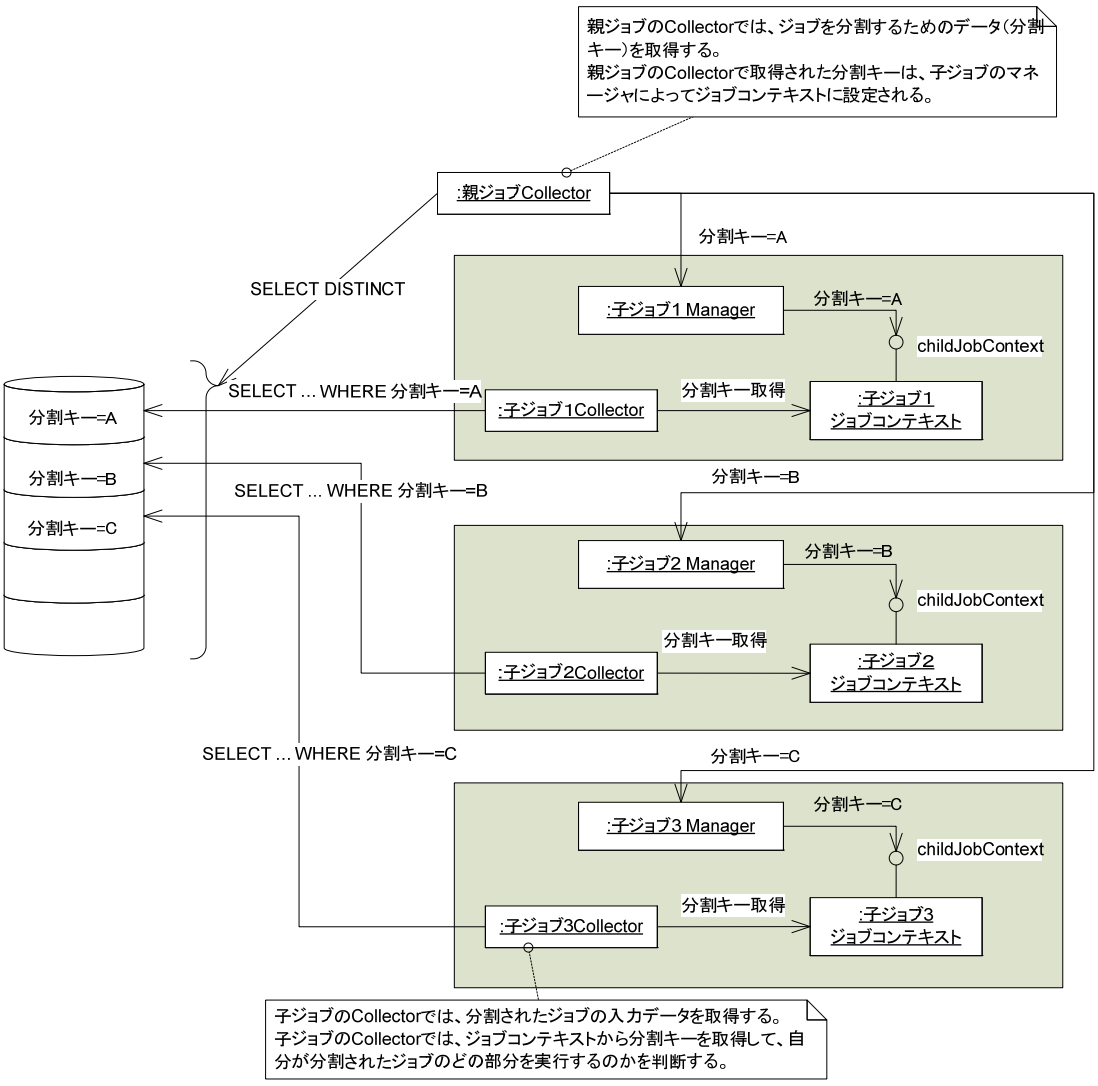
- フレームワークで用意している Collector デフォルト実装の種類
  フレームワークは、対象データの種類に対応して以下のデフォルト実装を提供する。

項番	Collector のデフォルト実装	対象データの種類	概要
1	iBATIS データベース Collector jp.terasoluna.fw.batch.ibatissupport.IBatisDbCollectorImpl	データベース	『BB-01 データベースアクセス機能』で提供する iBATIS で実装された DAO を使って、データベースから対象データを取得する
2	ファイル Collector jp.terasoluna.fw.batch.standard.StandardFileCollector	CSV ファイル、固定長ファイル、可変長ファイル	『BC-01 ファイルアクセス機能』を用いて、CSV ファイル/固定長ファイル/可変長ファイルから対象データを取得する
3	リストプロパティ Collector jp.terasoluna.fw.batch.standard.ListPropertyCollector	ジョブ Bean 定義ファイルで設定されたプロパティ	ジョブ Bean 定義ファイルで Collector に設定されたリストプロパティから、対象データを取得する
4	文字列配列プロパティ Collector jp.terasoluna.fw.batch.standard.StringArrayPropertyCollector		ジョブ Bean 定義ファイルで Collector に設定された文字列配列プロパティから、対象データを取得する

- データの取得条件
  Collector 実装には、Collector インタフェースを通してジョブコンテキストが渡される。Collector 実装では、渡されたジョブコンテキストを対象データ取得条件（SQL の WHERE 句にバインドする等）として利用する。
  iBATIS データベース Collector では、ジョブコンテキストが iBATIS SQL 定義ファイルでの parameterClass として利用される。

- 分割ジョブでの対象データ取得
  分割ジョブでは、親ジョブ（分割を行うジョブ）と子ジョブ（分割されたそれぞれの部分を実行するジョブ）のそれぞれに対して対象データを取得するための Collector を設定する。
  親ジョブと子ジョブの Collector では、それぞれ以下のデータを取得する。

項番	ジョブの種類	Collector で取得するデータ	例
1	親ジョブ	ジョブを分割するためのデータ（分割キー）	都道府県毎にジョブを分割する場合には、都道府県コードを取得する、等
2	子ジョブ	分割されたジョブにおいて、特定の分割キーに紐づく対象データ	都道府県毎にジョブを分割する場合には、親ジョブで取得された都道府県コードごとにデータを取得する、等



- 対象データが 0 件であったときの扱い
  iBATIS データベース Collector、ファイル Collector において、対象データが 0 件であった場合、デフォルトの処理結果ハンドラではそのまま正常終了する。
  対象データが 0 件であった場合に、警告ログの出力などの処理を行う場合には、処理結果ハンドラを実装して設定する。詳細は、『BE-05 処理結果ハンドリング機能』を参照のこと。

- 入力ファイルが存在しない場合の扱い
  ファイル Collector において、指定された入力ファイルが存在しなかった場合、および指定された入力ファイルの読込み権限がない場合には例外として処理される。詳細は、『BH-01 例外ハンドリング機能』を参照のこと。

- リスタート時の処理
  リスタート時には、Collector において入力データを読み飛ばすための処理が必要である。リスタートであるかどうかは、ジョブコンテキストの「リスタート区分」を参照して判定することができる。リスタートである場合には、ジョブコンテキストからリスタートポイントを取得し、リスタートポイントまでデータを読み飛ばすための SQL 記述等が必要である。詳細は使用方法を参照のこと。
  ファイル Collector では、『BC-01 ファイルアクセス機能』が提供する FileLineIterator 実装で用意される読み飛ばし用のメソッド（sikp）を使って読み飛ばしを行うように実装する必要がある。

- CollectorDataHandler
  Collector で取得したデータは、CollectedDataHandler で処理される。チャンク化が必要な場合には、Collector ではなく CollectedDataHandler によって行われる。
  通常ジョブ、または分割ジョブの子ジョブでは、Collector で取得された入力データはチャンクにまとめられる。チャンク化は、チャンク作成を行う CollectedDataHanlder である Chunker によって行われる。
  分割ジョブの親ジョブに設定された Collector では、分割キーを取得するためチャンクは作成されない。親ジョブの Collector には、分割キーを処理するための CollectedDataHanlder である PartitionKeyHandler を設定する。

- 対象データの入力チェック機能
  対象データ取得機能では、取得したデータに対して入力チェックを行うことができる。ジョブ Bean 定義ファイルにおいて Collector を指定する際に、入力チェック用 Collector として設定するとことで入力チェックを行うことができる。入力チェック用 Collector では、CollectedDataHandler でチャンクを作成する前に Validator による入力チェックを行う。
  Validator によるチェック結果は入力チェック処理結果ハンドラに渡される。入力チェック処理結果ハンドラが True を返却した場合には該当データをチャンクへ追加し、False を返却した場合には該当データをチャンクへ追加せずスキップする。
  入力チェック処理結果ハンドラが例外をスローした場合には、Collector による対象データ取得処理が終了する。
  異常データがある場合 False を返却するか、例外を投げるかはジョブ Bean 定義ファイルにて指定することができる。(デフォルト設定は False を返却し、処理を継続する。)
  ※ 入力チェックについては TERASOLUNA Server Framework for Java (Rich 版) の『RF-02 入力チェック機能』(以下、『RF-02 入力チェック機能』) の機能説明書を参照のこと。利用方法については『RF-02 入力チェック機能』の『使用方法』を参照のこと。チェックルールについては『RF-02 入力チェック機能』の『入力チェックルール解説』を参照のこと。ただし、コントローラに関する説明についてはこの『BD-02 対象データ取得機能』と関係ないので参照しなくてもよい。
  ※ 入力チェックについては、単項目入力チェックと相関項目入力チェック (コーディングによる入力チェック) の 2 種類があるが、2 つを併用する場合は単項目入力チェックが優先されるように設定を行う必要がある。

- Collector での処理結果
  Collector は対象データの取得が全て終了すると、処理結果として CollectorResult を返却する。CollectorResult は、以下のリターンコードを持つ。

項番	リターンコード	概要
1	NORMAL_END	Collector での対象データの取得が正常に終了したことを示す。
2	ERROR_END	Collector での対象データの取得において、エラーが発生し異常終了したことを示す。

  CollectorResult は、全ての対象データの取得が行われてから返却される。
  CollectorResult は、『BE-05 処理結果ハンドリング機能』で記述されている処理結果ハンドラによって処理される。詳細は、『BE-05 処理結果ハンドリング機能』を参照のこと。

- 対象データ取得でエラーが発生した場合
  通常ジョブ、あるいは分割ジョブの子ジョブにおいて、対象データの取得でエラーが発生した場合には、CollectorResult として ERROR_END が返されるか、あるいは Collector から例外がスローされる。
  CollectorResult は、処理結果ハンドラによって処理される。デフォルトの処理結果ハンドラでは、ERROR_END のリターンコードが設定された CollectorResult が返却された場合には、読込みが成功していたチャンクの処理の終了を待ってジョブ（あるいは子ジョブ）が終了する。
  Collector から例外がスローされた場合には、例外は例外ハンドラによって処理される。デフォルトの例外ハンドラでは、上記と同様の処理が行われる。
  詳細は、『BE-05 処理結果ハンドリング機能』、および『BH-01 例外ハンドリング機能』を参照のこと。

- 分割キー取得でエラーが発生した場合
  分割ジョブの親ジョブにおいて、分割キー取得でエラーが発生した場合にも、CollectorResult として ERROR_END が返されるか、あるいは Collector から例外がスローされる。
  これらは、処理対象データ取得でエラーが発生した場合と同様に、正常に読み込まれた分割キーの処理（すなわちその分割キーに対する子ジョブの処理）の終了を待って親ジョブが終了する。

- PostgreSQL 利用時に大量データ取得する際の留意点
  PostgreSQL 利用時に大量データを取得する際には、メモリ不足とならないように適切に設定を行う必要がある。PostgreSQL の JDBC ドライバでは、デフォルトの動作では SELECT 実行して ResultSet を返却する際に SELECT 結果を全件フェッチして Java のメモリに載せてしまう。SELECT 結果の件数が大量である場合には、メモリを大量に使用する。
  PostgreSQL の JDBC ドライバで、上記のような動作をさせないためには以下のような点に留意して設定を行う必要がある。

項番	設定項目	設定を行うファイル	設定内容
1	フェッチサイズ	iBATIS マッピング定義ファイル	<select>タグの属性"fetchSize"が指定されていること。
2	オートコミットモード	dataAccessContext-batch.xml	オートコミットモードが false に設定されていること。

  PostgreSQL の JDBC ドライバの仕様の詳細は、PostgreSQL JDBC ドライバのドキュメントを参照のこと。

■ 使用方法

◆ コーディングポイント

- iBATIS データベース Collector の設定

iBATIS データベース Collector では、以下のプロパティを設定する。

項番	プロパティ	データ型	必須	設定内容
1	sql	String	○	iBATIS の SQL 定義ファイルで定義された SQL ID を設定する
2	queryDAO	QueryDAO (『BB-01 データベースアクセス機能』で規定するインタフェース)	○	『BB-01 データベースアクセス機能』の Spring+iBATIS 連携機能を利用したデフォルト実装を設定する

➢ ジョブ Bean 定義ファイルの実装例

```
<!--ジョブの定義-->
  ...
<bean id="collector"
    parent="IBatisDbChunkCollector">
     <property name="sql" value=" sq0001"/>
</bean>
   ...
<bean id="jobContext"
 class="jp.terasoluna.xxx.SampleJobContext"/>
   ...
```

ID が"collector"の Bean に、iBATIS データベース Collector を指定する

iBATIS データベース Collector のプロパティを設定する

iBATIS データベース Collector では、パラメータとしてジョブコンテキストが利用できる

データベース Collector では、パラメータオブジェクトはジョブコンテキストとなる。

➢ SQL 定義ファイルの記述例

```
<select id="sq0001"
        parameterClass="jp.terasoluna.xxx.SampleJobContext"
          resultClass="jp.terasoluna.xxx.bean.XXXInputData" fetchSize="50">
SELECT INVOICE_NO, AMOUNT, NAME, ADDRESS, DATE FROM INVOICE WHERE DATE < #unyuHdk#
</select>
```

フェッチ行数の設定。PostgreSQL で大量のデータを取得する際は適切なサイズを指定すること。

ジョブコンテキストのプロパティを、パラメータとして利用できる。

- ビジネスロジックの実装例

```
public class XXXBLogic implements BLogic<XXXInputData, SampleJobContext> {
    private QueryDAO queryDAO;
   private UpdateDAO updateDAO;

 public BLogicResult execute(XXXInputData inputData,
     SampleJobContext jobContext) {
       ...
  }
}
```

iBATIS データベース Collector の resultClass で指定したクラスをビジネスロジックの入力として指定する

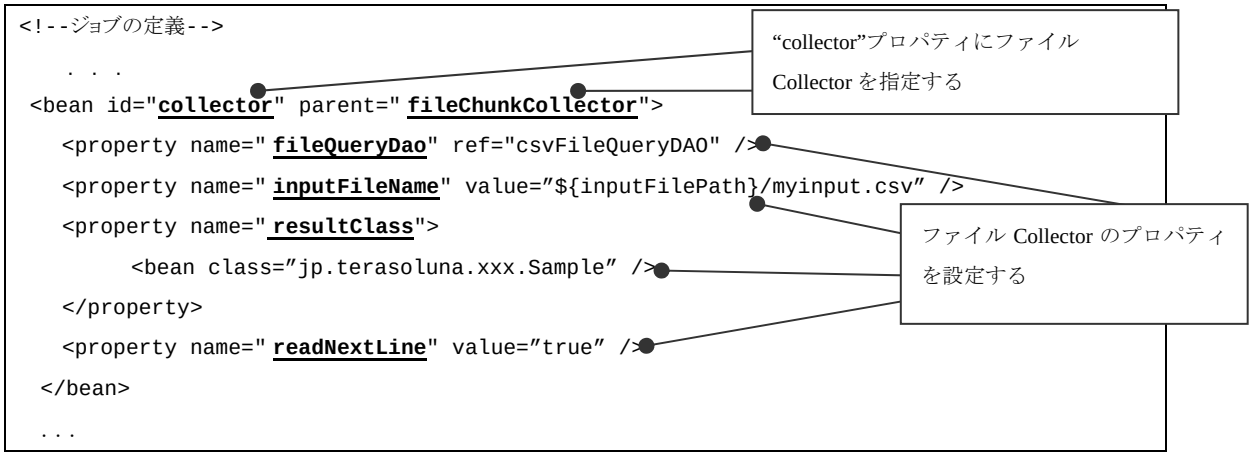
- ファイル Collector の設定

ファイル Collector では、以下のプロパティを設定する。

項番	プロパティ	データ型	必須	設定内容
1	fileQueryDAO	FileQueryDAO (『BC-01 ファイルアクセス機能』で規定するインタフェース)	○	『BC-01 ファイルアクセス機能』で提供する FileQueryDAO の実装クラスを設定する。 CSV ファイル/固定長ファイル/可変長ファイルのそれぞれに対応したクラスを設定する。

➢ ジョブ Bean 定義ファイルの実装例（CSV ファイルの場合）

```
<!--ジョブの定義-->
   . . .
 <bean id="collector" parent="fileChunkCollector">
   <property name="fileQueryDao" ref="csvFileQueryDAO" />
   <property name="inputFileName" value="${inputFilePath}/myinput.csv" />
   <property name="resultClass">
        <bean class="jp.terasoluna.xxx.Sample" />
   </property>
   <property name="readNextLine" value="true" />
 </bean>
 ...
```



- リストプロパティ Collector の設定

リストプロパティ Collector では、以下のプロパティを設定する。

項番	プロパティ	データ型	必須	設定内容
1	dataList	List	○	対象データをリスト形式で設定する。

➢ ジョブ Bean 定義ファイルの実装例

```
<!--ジョブの定義-->
   ...
<bean id="collector"
   parent="listProperty PartitionKeyCollector">
        <property name="dataList">
          <list>
            <bean class="jp.terasoluna.xxx.Prefecture">
              <property name="code" value="01" />
              <property name="name" value="北海道" />
            </bean>
            <bean class="jp.terasoluna.xxx.Prefecture">
              <property name="code" value="02" />
              <property name="name"  value="青森" />
            </bean>
       ...
          </list>
        </property>
</bean>
```

"collector"プロパティにリストプロパティ Collector を指定する

dataList プロパティを設定する

リストの要素には任意のクラスが設定できる

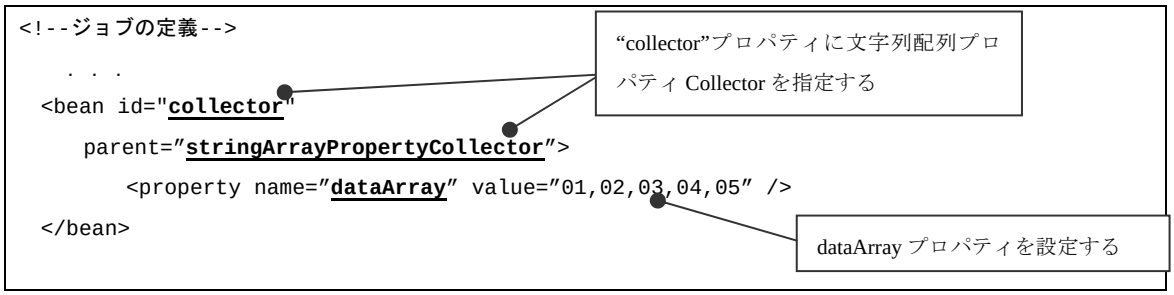
リストプロパティ Collector は、分割ジョブでの親ジョブにおいて分割キーを取得する際に利用することができる。

- 文字列配列プロパティ Collector の設定

文字列配列プロパティ Collector では、以下のプロパティを設定する。

項番	プロパティ	データ型	必須	設定内容
1	dataArray	String 配列	○	対象データをカンマ区切りで設定する。

➢ ジョブ Bean 定義ファイルの実装例



文字列配列プロパティ Collector は、分割ジョブでの親ジョブにおいて分割キーを取得する際に利用することができる。

- 分割ジョブで使用できる Collector のデフォルト実装

分割ジョブの親ジョブ、子ジョブにて利用できる Collector のデフォルト実装は以下のとおりである。

項番	Collector のデフォルト実装	親ジョブ (Bean 定義名)	子ジョブ (Bean 定義名)
1	iBATIS データベース Collector	○	○
		IBatisDbPartitionKeyCollector	IBatisDbChunkCollector
2	ファイル Collector	○※	×
		filePartitionKeyCollector	fileChunkCollector
3	リストプロパティ Collector	○※	×
		listPropertyPartitionKeyCollector	listPropertyChunkCollector
4	文字列配列プロパティ Collector	○	×
		stringArrayPropertyPartitionKey Collector	stringArrayPropertyChunk Collector

※ ジョブコンテキストの#getChildJobContext()をオーバーライドしない場合は、取得データ型が String 型に変換可能であるという制限がある。

- ジョブ Bean 定義ファイルの実装例

```
<!--ジョブの定義-->
   ・・・
 <bean id="partitionkeyCollector"
 parent="stringArrayPropertyPartitionKeyCollector"
       <property name="dataArray" value="01,02,03,04,05" />
 </bean>

 <bean id="collector"
    parent="IBatisDbChunkCollector">
       <property name="sql" value="childSql" />
 </bean>
```

“partitionkeyCollector”プロパティに親ジョブ用の文字列配列プロパティ Collector を指定する

“collector”プロパティに子ジョブ用の iBATIS データベース Collector を指定する

- 入力チェック用 Collector の設定

iBATIS データベース Collecotr、ファイル Collector、リストプロパティ Collector、および文字列配列プロパティ Collector のそれぞれに対して、入力チェック機能を追加された Collector がフレームワークから提供されている。
対象データ取得処理において、入力チェックを行う際にはジョブ Bean 定義ファイルで入力チェック機能を追加された Collector を指定する。

- Collector の実装例

```
<!--ジョブの定義-->
    . . .
<bean name="collector" parent="validationIBatisDbChunkCollector">
      <property name="sql" value=”UC-CS.SELECT003” />
</bean>
   . . .
```

入力チェック用 Collector を指定する

入力チェック機能を追加された Collector を利用する際には、それぞれの基となっている Collector のプロパティに加えて、以下を設定する。

- 入力チェック用 XML 設定ファイルのパスの設定

入力チェック用 XML 設定ファイルのパスを指定するため、ジョブ Bean 定義ファイルに validationConfigLocations として List を登録する。validationConfigLocations として Spring フレームワークの ListFactoryBean を登録し、以下のプロパティを設定する。

項番	プロパティ	データ型	必須	設定内容
1	sourceList	List	○	入力チェック用 xml 設定ファイルを設定

- validationConfigLocations の実装例

```
<!--ジョブの定義-->
    . . .
<util:list id="validationConfigLocations">
    <value>/common/validator-rules.xml</value>
    <value>/common/validator-rules-ex.xml</value>
    <value>/sample/UC-CS-002/validation.xml</value>
</bean>
   . . .
```

入力チェック用 xml 設定ファイルを設定

➢ 入力チェック処理結果ハンドラの設定

入力チェック処理結果ハンドラのデフォルト動作ではなく、ジョブ固有の動作を行わせたいときには、ジョブ Bean 定義ファイルにおいて入力チェック用 validationResultHandler を上書きして定義する。

validationResultHandler には、フレームワークが規定するインタフェース（jp.terasoluna.fw.batch.validation.ValidationExecutorFactory）を実装したクラスを設定する。

入力チェック用 validationResultHandler では、以下のプロパティを設定する。

項番	プロパティ	データ型	必須	設定内容
1	errorContinueFlg	boolean		異常データがある場合も処理を継続する場合『true』を設定
1	messageAccessor	MessageAccessor		メッセージ取得クラス（errorContinueFlg を true にした場合は必須）

➢ validationResultHandler の実装例

```
<!--ジョブの定義-->
    . . .
 <bean id="validationResultHandler"
       class="jp.terasoluna.fw.batch.validation.StandardValidationResultHandler">
    <property name="errorContinueFlg" value="true"/>
    <property name="messageAccessor" ref="messageAccessor"/>
 </bean>
    . . .
```

使用するハンドラの設定

フレームワークが提供するハンドラを使用する場合は設定が必要
errorContinueFlg を true に設定した場合は異常があるデータをスキップし（チャンクに含めず）、処理を継続する。

➢ 単項目入力チェックと相関項目入力チェックの併用設定

単項目入力チェックが相関項目入力チェックよりも優先して実行されるように設定をする。

➢ 単項目チェックと相関チェックの併用設定例

```
<!--ジョブの定義-->
    . . .
<!-- コレクターの設定 -->
<bean id="collector"
        class="jp.terasoluna.fw.batch.standard.StandardFileCollector">
  <!-- 単項目入力チェック設定 -->
   <property name="collectedDataHandlerFactory">
     <bean class="jp.terasoluna.fw.batch.validation.ValidationExecutorFactory">
       <property name="validator"><bean parent="validator" /></property>
       <property name="validationResultHandler"><bean parent="validationResultHandler" /></property>
        <!-- 相関項目入力チェック設定 -->
        <property name="collectedDataHandlerFactory">
          <bean class="jp.terasoluna.fw.batch.validation.ValidationExecutorFactory">
          <property name="validator"><bean parent="sampleValidator" /></property>
          <property name="validationResultHandler"><bean parent="validationResultHandler" /></property>
            <!-- チャンク生成の設定 -->
            <property name="collectedDataHandlerFactory">
              <bean class="jp.terasoluna.fw.batch.standard.ChunkerFactory">
                <property name="chunkSize" ref="chunkSize" />
              </bean>
            </property>
          </bean>
       </property>
     </bean>
   </property>
   <!-- コレクタ自身の設定 -->
   <property name="fileQueryDao" ref="csvFileQueryDAO" />
   <property name="inputFileName" value="inputfile/nyukindata.csv"/>
   <property name="resultClass">
     <bean class="jp.terasoluna.batch.tutorial.uc0001.JB0002Data"/>
   </property>
   <property name="readNextLine" value="false" />
</bean>

<!-- 相関項目入力チェック -->
<bean id="sampleValidator" class="jp.terasoluna.batch.tutorial.uc0001.jb0004.SampleMultiFieldsValidator">
   <property name="validatorFactory"><bean parent="validatorFactory" /></property>
   <property name="useFullyQualifiedClassName" value="false" />
</bean>
    . . .
```

使用する入力チェックを順番にネストして”collectedDataHandlerFactory”に指定する

➢ リスタート時のデータ読み飛し実装例

フレームワークが提供するデータアクセス機能を使う場合は、リスタートポイント以前のデータを読み飛ばすように記述した SQL 文を設定する。

```
<!-- sqlMapの定義 postgreSqlの場合-->
    . . .
<select id="SELECT001" resultClass="be01BankTransactionData"
     parameterClass="jp.terasoluna.batch.sample.be01.SampleJobContext" fetchSize="100">
 SELECT BID,TID,AID,DELTA FROM BANKTRANSACTIONDATA ORDER BY AID OFFSET #restartPoint#
</select>
    . . .
```

◆ 拡張ポイント

- Collector 実装の追加
  プロジェクトの要求により、Collector インタフェースの実装クラスを追加することができる。

■ 関連機能

- 『BB-01 データベースアクセス機能』
- 『BC-01 ファイルアクセス機能』
- 『BE-05 処理結果ハンドリング機能』
- 『BH-01 例外ハンドリング機能』
- TERASOLUNA Server Framework for Java（Rich 版）『RF-02 入力チェック機能』

■ 使用例

- 機能網羅サンプル(functionsample-batch)
- チュートリアル(tutorial-batch)

■ 備考

- なし。

BD-03 コントロールブレイク機能

■ 概要

◆ 機能概要

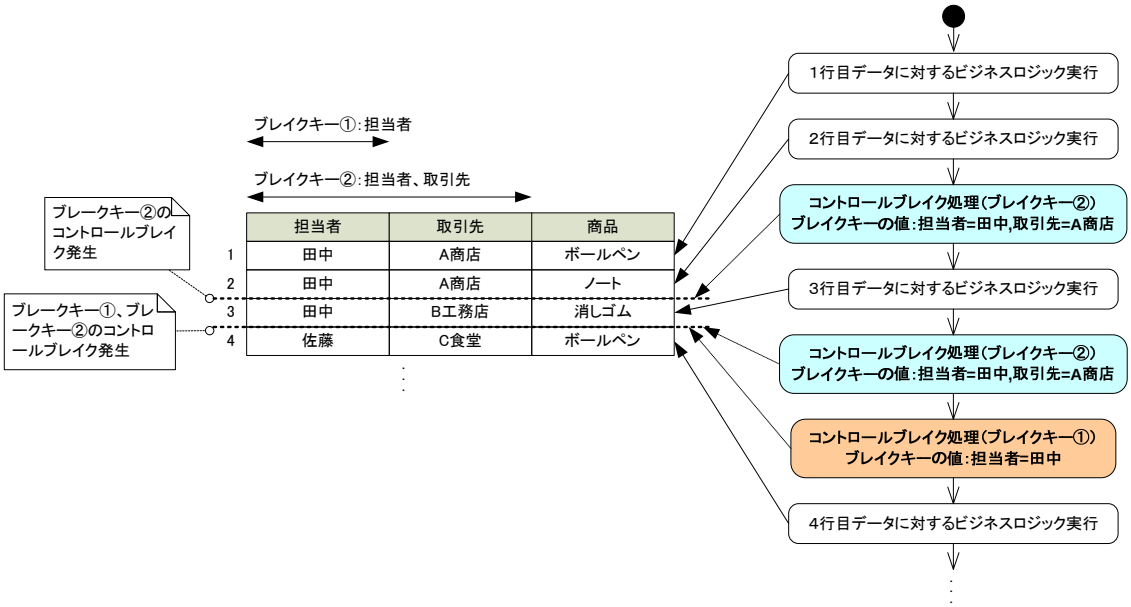
- コントロールブレイク機能では、対象データがコントロールブレイクした際に、コントロールブレイクハンドラを起動する機能を提供する。
  - コントロールブレイク処理とは、ある項目をキーとして、キーが変わるまで集計したり見出しを追加したりする処理である。キーブレイク処理とも呼ばれる。

- TERASOLUNA-Batch では、以下の３種類のコントロールブレイクを規定している。

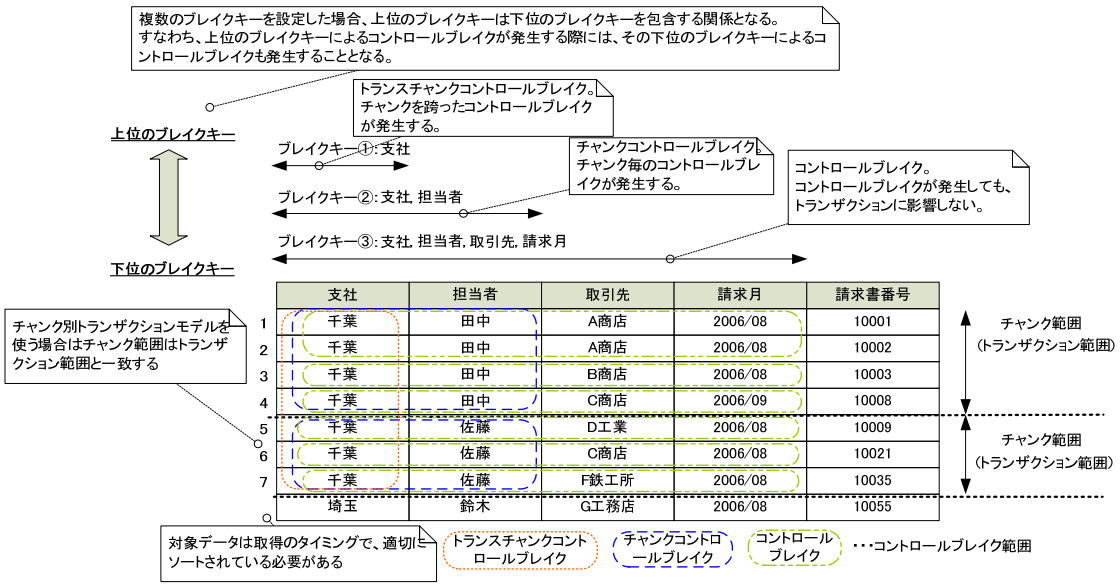
項番	コントロールブレイクの種類	チャンクとの関係	トランザクション範囲との関係	定義可能なコントロールブレイクの数
1	チャンクコントロールブレイク	チャンクの切れ目においてコントロールブレイクが発生する。	チャンク別トランザクションモデル（ブレイク）雛形を使用すればトランザクション境界毎にブレイク処理が発生する。	ジョブに対して、一つのみ定義可能
2	トランスチャンクコントロールブレイク	チャンクの切れ目を跨ってコントロールブレイクが発生する。	チャンク別トランザクションモデル（ブレイク）雛形を使用すればトランザクション境界を跨ったブレイク処理が発生する。	ジョブに対して、複数定義可能。ただし、チャンクコントロールブレイクの設定が無い場合は定義できない。
3	コントロールブレイク	チャンク内の処理においてチャンクとは関係なく処理する入力データ単位でコントロールブレイクを発生させることができる。	チャンク内の処理であるためトランザクション境界と関係なくコントロールブレイク処理が起動される。	ジョブに対して、複数定義可能。

◆ 概念図

- コントロールブレイク処理の概要
  ビジネスロジックが実行されると、次のデータに対してビジネスロジックが実行される前にフレームワークによってコントロールブレイク発生の判定が行われる。
  コントロールブレイク発生の判定は、アプリケーション開発者が設定したブレイクキーに従って行われる。
  コントロールブレイクが発生していた場合には、当該ブレイクキーに対応するコントロールブレイク処理が起動される。



- チャンクコントロールブレイク、トランスチャンクコントロールブレイクとコントロールブレイクの例



	支社	担当者	取引先	請求月	請求書番号
1	千葉	田中	A商店	2006/08	10001
2	千葉	田中	A商店	2006/08	10002
3	千葉	田中	B商店	2006/08	10003
4	千葉	田中	C商店	2006/09	10008
5	千葉	佐藤	D工業	2006/08	10009
6	千葉	佐藤	C商店	2006/08	10021
7	千葉	佐藤	F鉄工所	2006/08	10035
	埼玉	鈴木	G工務店	2006/08	10055

➢ 処理の流れ

ステップ		処理内容
1	1	1行目の対象データに対するビジネスロジック処理が実行される
2	1	2行目の対象データに対するビジネスロジック処理が実行される
	2	ブレイクキー③に対応するコントロールブレイク処理が実行される
3	1	3行目の対象データに対するビジネスロジック処理が実行される
	2	ブレイクキー③に対応するコントロールブレイク処理が実行される
4	1	4行目の対象データに対するビジネスロジック処理が実行される
	2	ブレイクキー③に対応するコントロールブレイク処理が実行される
	3	ブレイクキー②に対応するコントロールブレイク処理が実行される
	4	チャンクト別ランザクションモデル(ブレイク)を使う場合はトランザクション境界がチャンク単位になるため、コミットが行なわれる
5	1	5行目の対象データに対するビジネスロジック処理が実行される
	2	ブレイクキー③に対応するコントロールブレイク処理が実行される
6	1	6行目の対象データに対するビジネスロジック処理が実行される
	2	ブレイクキー③に対応するコントロールブレイク処理が実行される
7	1	7行目の対象データに対するビジネスロジック処理が実行される
	2	ブレイクキー③に対応するコントロールブレイク処理が実行される
	3	ブレイクキー②に対応するコントロールブレイク処理が実行される
	4	ブレイクキー①に対応するコントロールブレイク処理が実行される
	5	チャンク別トランザクションモデル(ブレイク)を使う場合はトランザクション境界がチャンク単位になるため、コミットが行なわれる

◆ 解説

- コントロールブレイクの種類とトランザクション
  フレームワークでは、以下の３種類のコントロールブレイクを規定する。
  それぞれのコントロールブレイクハンドラは、ジョブ Bean 定義ファイルの対応するプロパティで設定を行う。

項番	コントロールブレイクの種類	概要
1	チャンクコントロールブレイク	チャンクの切れ目でブレイク処理が発生する。チャンク別トランザクションモデル（ブレイク）雛形を使用するとチャンクの切れ目はトランザクション境界と一致するためトランザクション単位でブレイク処理が発生されることになる。 ジョブに対して一つのみ定義することができる。
2	トランスチャンクコントロールブレイク	チャンクを跨ったチャンクの切れ目でブレイク処理が発生する。チャンク別トランザクションモデル（ブレイク）雛形を使用するとトランザクション範囲を超えたブレイク処理になる。 ジョブに対しては複数定義することはできるが、チャンクコントロールブレイク定義が無い場合はトランスチャンクコントロールブレイクを定義することはできない。また、チャンクコントロールブレイクの設定内容を包含した定義にする必要がある。
3	コントロールブレイク	チャンク内の `BLogic` で処理する入力データ毎にブレイクキーを確認し、ブレイク処理を実行する。チャンク内の `BLogic` で処理する入力データ単位のブレイク処理であるためコントロールブレイクが発生してもトランザクション境界には影響しない。 チャンクコントロールブレイク定義有無にかかわらずジョブに対しては複数定義することができる。ただし、チャンクコントロールブレイク定義がある場合はチャンクコントロールブレイクの設定内容を包含した定義にする必要がある。

チャンクコントロールブレイクでは、チャンクを作成する際にチャンクサイズは参照されず、コントロールブレイクの範囲がチャンクとなるようにチャンク内のデータのサイズが調整される。ただし、チャンクコントロールブレイクの設定が無い場合は指定されたチャンクサイズを参照しチャンクを作成する。
トランスチャンクコントロールブレイクでは、チャンクコントロールブレイク用のチャンクを作成する際にチャンクを跨ったブレイク処理があるかを評価し、ブレイク処理情報をチャンクに追加する。追加された情報はチャンクの切れ目で評価結果に従ったブレイク処理を起動する。
コントロールブレイクでは、トランザクション範囲はコントロールブレイク範囲とは無関係に設定される。すなわち、コントロールブレイク範囲の中であっても、チャンクサイズの設定に従ってフレームワークによってトランザクション範囲が設定される。

➢ 包含関係について

フレームワークが規定する３種類のコントロールブレイクには、包含関係がある。
その包含関係と具体例は、以下のとおりである。

包含関係)
トランスチャンクコントロールブレイク⊆チャンクコントロールブレイク⊆コントロールブレイク

具体例)
｛支社｝⊆｛支社、担当者｝⊆｛支社、担当者、取引先、請求月｝

- コントロールブレイクハンドラのインタフェース
  コントロールブレイクハンドラはフレームワークで規定する以下のインタフェースを実装して作成する。

項番	インタフェース名	概要
1	jp.terasoluna.fw.batch.openapi.ControlBreakHandler	コントロールブレイクハンドラが実装するインタフェース

  フレームワークは、ジョブ Bean 定義ファイルのコントロールブレイクの設定により、コントロールブレイク発生の判定を行い、ControlBreakHandler インタフェースを通してコントロールブレイクハンドラを起動する。

- コントロールブレイクハンドラの入力
  コントロールブレイクでは、以下の入力パラメータが ControlBreakHandler インタフェースを通して渡される。

項番	入力パラメータ	概要
1	ブレイクキー値マップ	コントロールブレイクが発生した際のブレイクキーの値。 ブレイクキーが複数の項目である場合には、それらの値が List で渡される。たとえば、「取引先番号」と「請求書番号」でブレイクキーとして設定されていた場合には、コントロールブレイク時の「取引先番号」の値と「請求書番号」の値がマップ形式で渡される。
2	ジョブコンテキスト	ジョブのジョブコンテキスト。 コントロールブレイク処理でサマリなどを取る場合には、ジョブコンテキストからサマリを取れるようにビジネスロジックの中で順次加算するなどの処理を行う必要がある。一方、コントロールブレイクハンドラでは次回コントロールブレイク時にもサマリが正しく取得できるように、ジョブコンテキストのサマリ結果をクリアするなどの処理が必要となる。

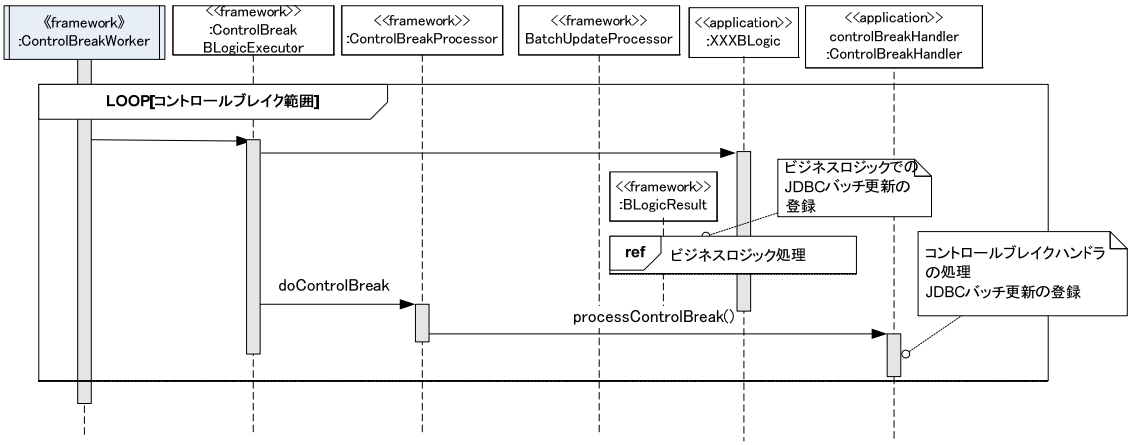
- コントロールブレイクハンドラの起動条件
  コントロールブレイクハンドラは、以下の何れかの条件が成立するときに起動される。

項番	コントロールブレイクハンドラの起動条件
1	コントロールブレイクハンドラに対応するコントロールブレイクが発生した。
2	対象データの最後に到達した。

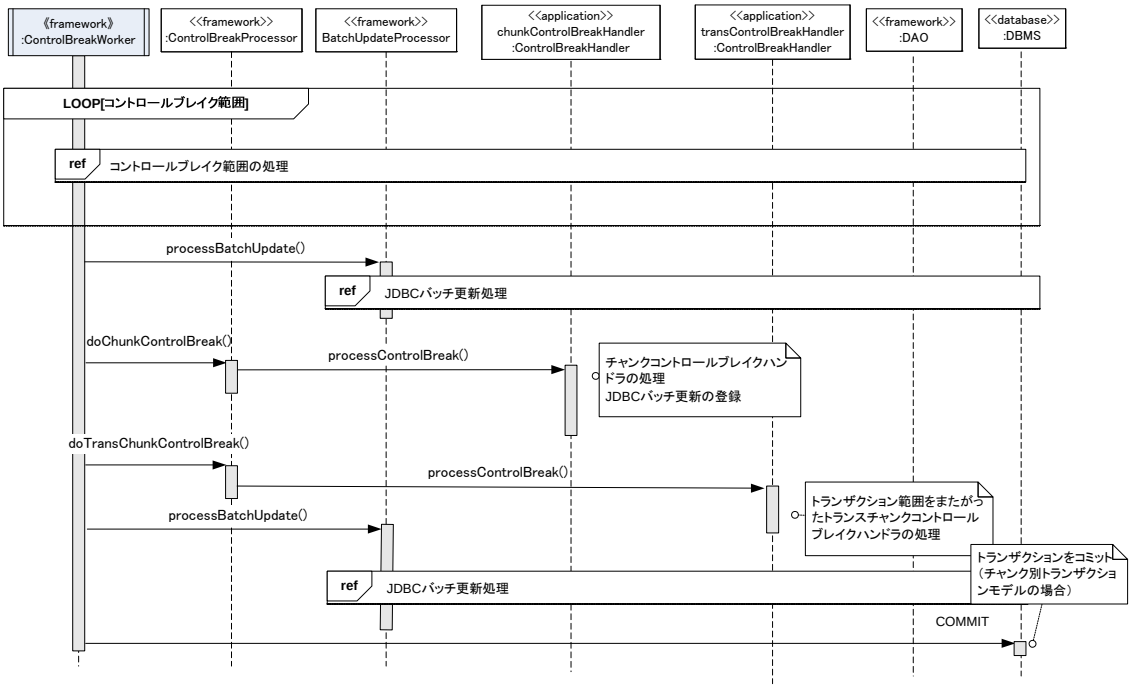
  コントロールブレイク範囲のビジネスロジック（あるいは処理結果ハンドラ、例外ハンドラ）で、NORMAL_END 又は ERROR_END が返却された場合には、コントロールブレイクハンドラは起動されない。また、コントロールブレイクが複数定義されていて、下位のコントロールブレイクハンドラが NORMAL_END 又は ERROR_END を返却した場合でも当該コントロールブレイクハンドラは起動されない。
  対象データの最後に到達した場合、及び最終チャンクに到達した場合には、全てのコントロールブレイクが発生したものとして扱われる。

- 一つのブレイクキーに対するコントロールブレイクハンドラの設定数
  一つのブレイクキーに対して一つのコントロールブレイクハンドラのみ設定することができる。ただし、チャンクコントロールブレイクに対するコントロールブレイクハンドラ、トランスチャンクコントロールブレイクに対するコントロールブレイクハンドラ、コントロールブレイクに対するコントロールブレイクハンドラは、同一のブレイクキーに対して設定することができる。

- コントロールブレイクハンドラの起動タイミング
  コントロールブレイクに対するコントロールブレイクハンドラは、コントロールブレイク発生時にコントロールブレイク範囲のビジネスロジックが起動された後で直ちに起動される。したがって、そのジョブがチャンク別トランザクションモデル(ブレイク)の場合に、たまたまチャンクの切れ目でコントロールブレイクハンドラが起動された場合であっても、そのチャンク分の JDBC バッチ更新が起動される前にコントロールブレイクハンドラが起動される。



チャンクコントロールブレイクに対するコントロールブレイクハンドラは、チャンク分の JDBC バッチ更新が実行された後、かつトランザクションがコミットされる前に起動される。



また、チャンクコントロールブレイク、トランスチャンクコントロールブレイクとコントロールブレイクが同一ジョブの同一ブレイクキーで設定されていた場合には、コントロールブレイクに対するコントロールブレイクハンドラが起動され

てから、チャンク分の JDBC バッチ更新処理が行われ、その後チャンクコントロールブレイクに対するコントロールブレイクハンドラ起動後、トランスチャンクコントロールブレイクに対するコントロールブレイクハンドラが起動される。
JDBC バッチ更新処理については、『BB-01 データベースアクセス機能』を参照のこと。

- 複数のコントロールブレイクが存在する場合の起動順序
  複数のコントロールブレイクが発生する場合には、下位のコントロールブレイクハンドラから順次起動される。
  たとえば、「支社コード」および「支社コード、担当者」でそれぞれにコントロールブレイクが定義されている場合では、「支社コード」の値が変わったときには、「支社コード、担当者」のコントロールブレイク、「支社コード」のコントロールブレイクの順に起動される。

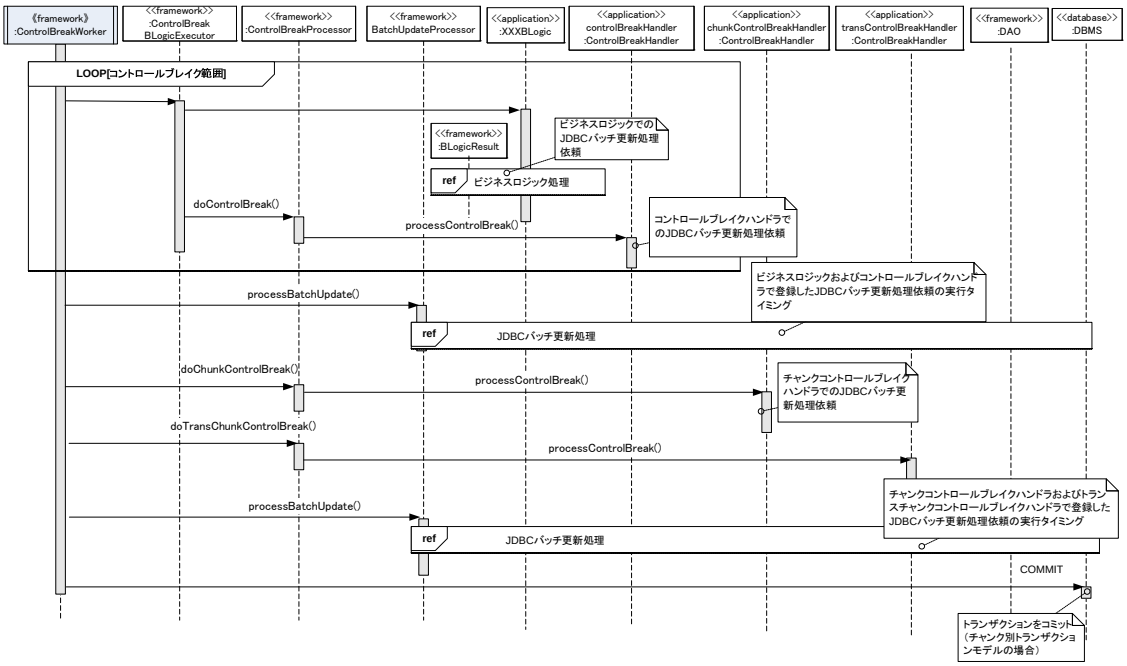
- 分割キーのコントロールブレイク
  分割キーに対しては、コントロールブレイクを定義することはできない。つまり、分割ジョブの親ジョブにはコントロールブレイクを定義することはきない。
  通常ジョブ、および分割キーの子ジョブに対してはそれぞれコントロールブレイクを定義することができる。
  ※分割ジョブについては、フレームワークでは雛形提供していない。分割ジョブの使用方法については、機能網羅サンプル・チュートリアルを参照のこと。

- コントロールブレイクハンドラの返却値
  コントロールブレイクハンドラでは、ビジネスロジックと同じく BLogicResult を返却する。また、BLogicResult にはリターンコードを設定して返却する。

項番	リターンコード	概要
1	NORMAL_CONTINUE	処理を継続する。 返却された NORMAL_CONTINUE は、JobResult が保持するブレイクキー毎の NORMAL_CONTINUE 件数に加算される。
2	NORMAL_END	(実行中のトランザクションがあれば)トランザクションをコミットし、ジョブを終了する。 コントロールブレイクハンドラが NORMAL_END を返却した場合には、上位のコントロールブレイクハンドラが存在したとしても起動されない。
3	ERROR_CONTINUE	処理を継続する。 返却された ERROR_CONTINUE は、JobResult が保持するブレイクキー毎の ERROR_CONTINUE 件数に加算される。
4	ERROR_END	(実行中のトランザクションがあれば)トランザクションをロールバックし、ジョブを終了する。 そのコントロールブレイクに対して複数のコントロールブレイクハンドラが起動される条件であっても、下位のコントロールブレイクハンドラが ERROR_END を返却した場合、上位のコントロールブレイクハンドラは起動されない。 コントロールブレイクハンドラが ERROR_END を返した場合に、同一のブレイクキーでトランザクショナルコントロールブレイクハンドラが設定されていても、そのトランザクショナルコントロールブレイクハンドラは起動されない。

- コントロールブレイクハンドラからの JDBC バッチ更新の登録
  コントロールブレイクハンドラでは、返却する BLogicResult に JDBC バッチ更新処理依頼を登録することができる。コントロールブレイクハンドラから登録した JDBC バッチ更新処理依頼は、以下のタイミングで実行される。

項番	コントロールブレイクの種類	コントロールブレイクハンドラで登録した JDBC バッチ更新処理依頼の実行タイミング
1	チャンクコントロールブレイク	コントロールブレイクハンドラが実行された後、トランザクションがコミットされる前に実行される。
2	トランスチャンクコントロールブレイク	コントロールブレイクハンドラが実行された後、トランザクションがコミットされる前に実行される。
3	コントロールブレイク	コントロールブレイクハンドラが含まれるチャンクの処理が終わったタイミングで実行される。



- コントロールブレイクハンドラでの例外処理
  コントロールブレイクハンドラから、フレームワークに例外がスローされてきた場合の処理については、『BH-01 例外ハンドリング機能』を参照のこと。

■ 使用方法

◆ コーディングポイント

- ジョブ Bean 定義ファイルの設定
  ジョブ Bean 定義ファイルでは、コントロールブレイク、およびコントロールブレイクハンドラを設定することができる。
  - ジョブ Bean 定義ファイルの設定例

```
...
<import resource="classpath:/template/ChunkTransactionForControlBreakBean.xml"/>
...
<bean id="collector" parent="controlBreakIBatisDbChunkCollector">
...
<util:list id="controlBreakDefItemList">
       <bean lass="jp.terasoluna.fw.batch.controlbreak.ControlBreakDefItem">
          <property name="breakKey">
            <list>
               <value>branchCd</value>
               <value>tantosyaId</value>
               <value>custName</value>
            </list>
          </property>
          <property name="contorlBreakHandler">
            <bean class="jp.terasoluna……XXXControlBreakHandler"/>
          </property>
       </bean>
</util:list>
<bean id="chunkControlBreakDefItem"
       class="jp.terasoluna.fw.batch.controlbreak.ControlBreakDefItem">
    <property name="breakKey">
       <list>
          <value>branchCd</value>
          <value>tantosyaId</value>
       </list>
    </property>
    <property name="contorlBreakHandler">
       <bean class="jp.terasoluna……XXXControlBreakHandler"/>
     </property>
</bean>
```

コントロールブレイク用の雛形をインポートする。

コントロールブレイク用 Collector 設定。

コントロールブレイクの設定。

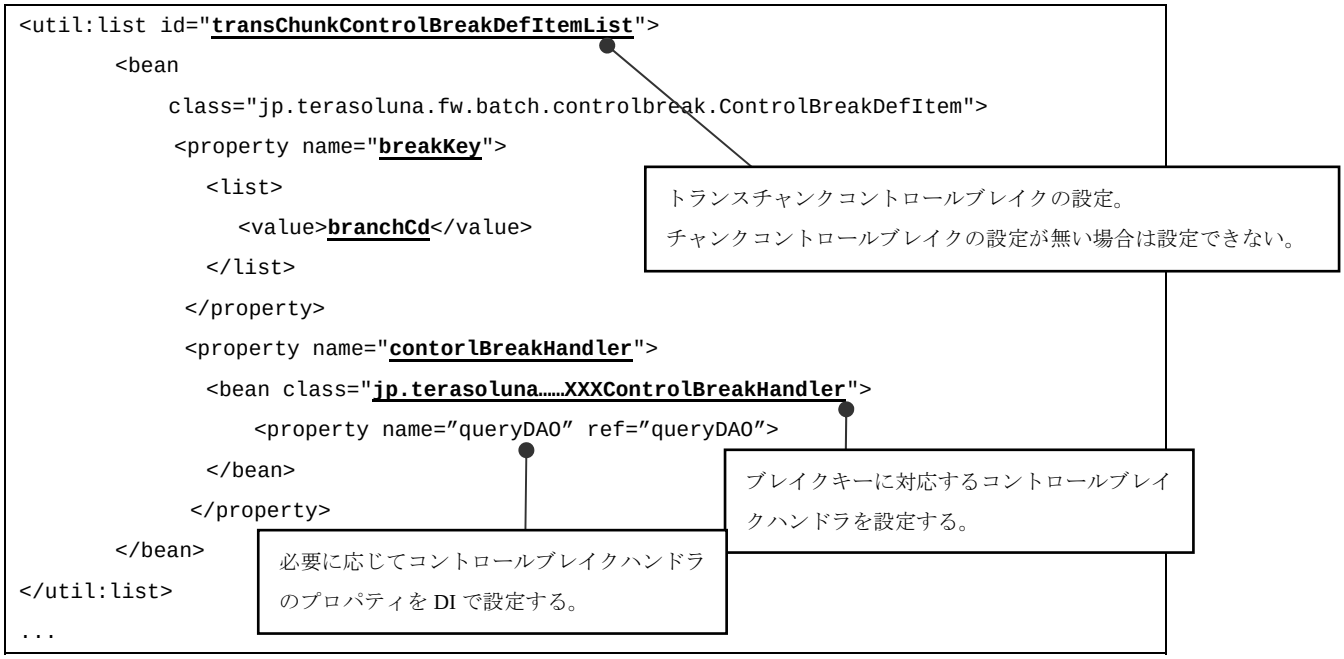
ブレイクキーには、ビジネスロジックの入力パラメータの属性名を指定する。

ブレイクキーに対応するコントロールブレイクハンドラを設定する。

チャンクコントロールブレイクの設定。

指定したキーにより、チャンクが作成される。

ブレイクキーに対応するコントロールブレイクハンドラを設定する。



➢ コントロールブレイクハンドラの実装例

```
pubic class TantosyaSummaryHandler implements ControlBreakHandler<MyJobContext> {
    private QueryDAO queryDAO;
    ...


    public BlogicResult handleControlBreak(Map<String, Object>breakKeyValueMap,
      MyJobContext jobContext) {


       ...
       String tantosyaId = (String) breakKyes.get(0);
       BigDecimal tantosyaBetsuKingaku = jobContext.getTantosyaBetsuKingaku();

        ...

       jobContext.clearTantosyaBetsuKingaku();

       return new BLogicResult(ReturnCode.NORMAL_CONTINUE);
    }
}
```

コントロールブレイクハンドラは、ControlBreakHandler を実装する。

コントロールブレイクハンドラでは、ジョブコンテキストなどからコントロールブレイクに必要なデータを取得する。

ジョブコンテキストでは大きなデータを保持しないように、不要なデータは削除する。

コントロールブレイクハンドラでは、BLogicResult をリターンする。

◆ 拡張ポイント

- 処理モデルの雛形作成。
  フレームワークで提供するブレイク処理用雛形はチャンク別トランザクションモデルをベールにした雛形であり、必要に応じてチャンク別ノートランザクションや分割ジョブ用のブレイク処理用雛形等を作成することができる。

■ 関連機能

- 『BA-01 トランザクション管理機能』
- 『BB-01 データベースアクセス機能』
- 『BH-01 例外ハンドリング機能』

■ 使用例

- 機能網羅サンプル(functionsample-batch)
- チュトリアル(tutorial-batch)

■ 備考

- なし。

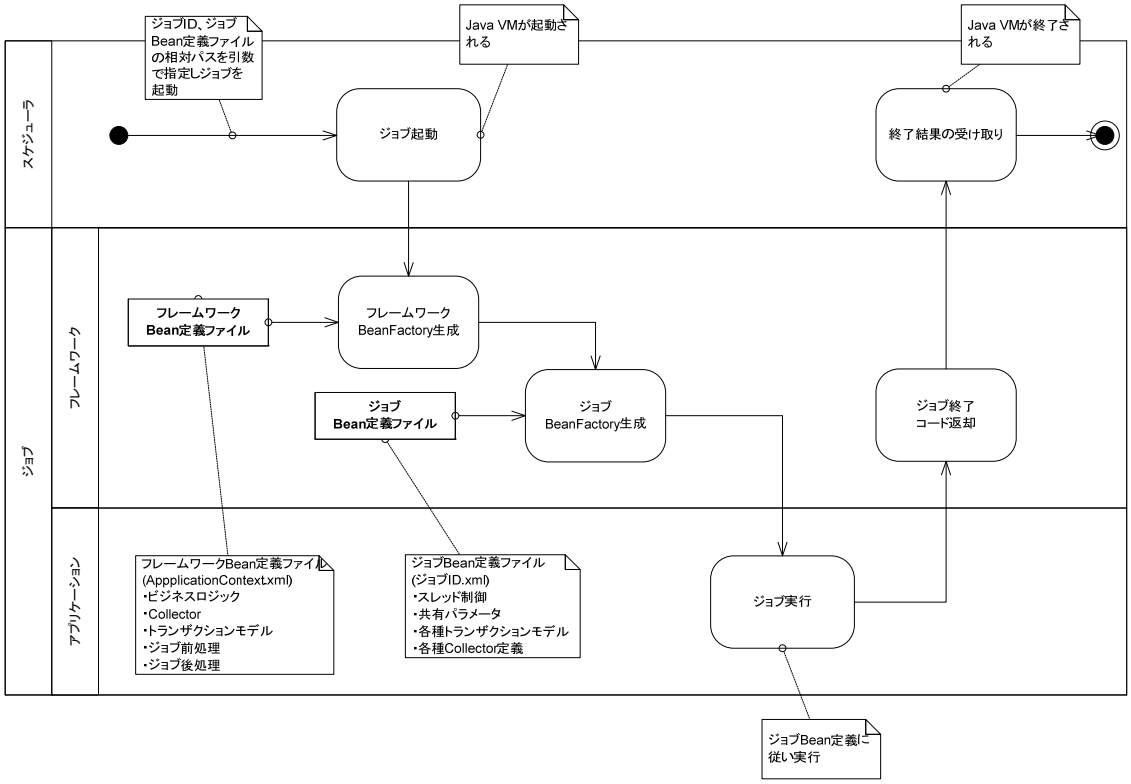
BE-01 同期型ジョブ起動機能

■ 概要

◆ 機能概要

- Java VM を起動し対象のジョブを実行する機能を提供する。
- 起動するジョブの設定はジョブ Bean 定義ファイル上で行う。
- バッチフレームワークにジョブ ID、ジョブ Bean 定義ファイルの相対パスを引数として指定することで対象のジョブが実行される。

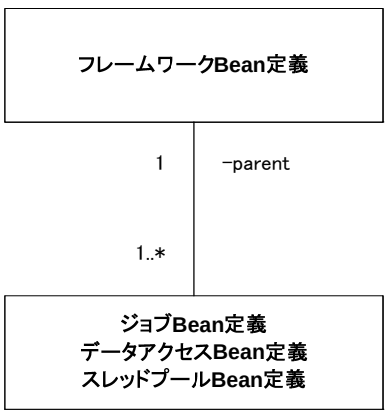
◆ 概念図



◆ 解説

- 同期型ジョブの起動と終了
  - Java VM の起動時、引数としてジョブ ID、ジョブ Bean 定義ファイルの相対パスを指定することで対象のジョブが実行される。
  - 起動されたフレームワークは初期化処理を行う。
    - フレームワーク Bean 定義ファイルを用いてフレームワーク BeanFactory の生成を行う。
    - ジョブ Bean 定義、データアクセス Bean 定義、スレッドプール Bean 定義ファイルを用いてジョブ BeanFactory の生成を行う。
    - 初期化処理に失敗した場合はエラーとして終了する。
    - フレームワーク Bean 定義ファイル、ジョブ Bean 定義ファイル、データアクセス Bean 定義ファイル、スレッドプール Bean 定義ファイルの詳細は『設定ファイル説明書』を参照のこと。
  - ジョブ Bean 定義ファイルの設定内容に従いジョブを実行する。
    - インポートしたジョブ Bean によりキー分割処理等を選択することができる。詳細は『BA-01 トランザクション管理機能』を参照のこと。
  - ジョブが完了したらジョブ終了コードを返却し、Java VM を終了する。
    - ジョブ終了コード返却に関する詳細な内容は『BD-01 ビジネスロジック実行機能』、『BE-05 処理結果ハンドリング機能』を参照のこと。

- フレームワーク BeanFactory とジョブ BeanFactory
  - フレームワーク BeanFactory はバッチフレームワーク全体で使われるベース Manager、Collector、処理結果ハンドラ等のリソース情報が格納されている Spring の BeanFactory である。
  - ジョブ BeanFactory はジョブ毎に実行するジョブ前処理、ジョブ後処理、Collector、ビジネスロジック等の設定情報が格納されている Spring の BeanFactory である。
  - ジョブ BeanFactory はフレームワーク BeanFactory を親 Bean として指定しジョブ定義を行うことでジョブ BeanFactory からフレームワーク BeanFactory の情報を参照することができる。



- データアクセス Bean 定義ファイルとスレッドプール Bean 定義ファイル
  - データアクセス Bean 定義ファイル
    データソース、iBATIS 用 SqlMapClient、DAO、トランザクションプロキシ等データアクセス用 Bean が定義されたファイルである。
    同期型ジョブ起動時には、ジョブ BeanFactory に Bean が登録される。ジョブ BeanFactory で iBATIS 設定ファイルが設定されている場合には、その設定にしたがって iBATIS が初期化される。データアクセスについての詳細は、『BB-01 データベースアクセス機能』を参照のこと。
  - スレッドプール Bean 定義ファイル
    ジョブ起動時に使用するスレッドプールを定義する Bean 定義ファイルである。
    同期型ジョブ起動時には、ジョブ BeanFactory に Bean が登録される。ジョブ Bean 定義ファイルの多重度の設定に従いスレッドプールを生成する。スレッド設定の詳細は『アーキテクチャ説明書』を参照のこと。

- ジョブ起動時の引数
  - 第１引数：起動するジョブ ID（必須）
    - 実行対象のジョブ ID を指定する。
    - 実行対象のジョブ Bean 名に該当する。
  - 第２引数：ジョブ Bean 定義ファイルの相対パス（必須）
    - 実行対象ジョブの情報が設定されているジョブ Bean 定義ファイルの相対パスを指定する。
      - ジョブ Bean 定義ファイルが「batchapps」フォルダに格納されている場合には、そのフォルダからの相対パスを指定する。「batchapps」フォルダの構成についてはコーディングポイントを参照のこと。
      - ジョブ Bean 定義ファイルは、クラスローダで読み込まれる。ジョブ Bean 定義ファイルが格納されたフォルダ、あるいは jar ファイルにクラスパスを通しておくこと。
    - 指定されたジョブ Bean 定義ファイルが存在しない場合はエラー終了する。
  - 第３引数以降：パラメータ値（任意）
    - 指定された値はジョブコンテキストに格納される。
    - 引数の前に「-p」を指定することで起動ジョブのジョブプロセス ID を指定することができる。ジョブプロセス ID の詳細は『BE-03 ジョブ実行管理機能』を参照のこと。
  - VM 引数：置換先の終了コード（任意）
    - アプリの終了コードが'1'の場合、-D オプションを利用する。
      - jp.terasoluna.fw.batch.jobExitCode1=（任意の終了コード）を設定して任意の終了コードに置換できる。-D オプションを設定しない場合は置換処理を行わずに終了コード'1'をそのまま返却する。また、任意の終了コードに int 型の範囲外の数値や文字列などを指定した場合は、例外が発生しログ出力を行ったうえで JavaVM が異常終了する。終了コードの説明については『BE-05 処理結果ハンドリング機能』を参照のこと。

- パラメータデータ初期化処理
  - ジョブ起動時に指定した引数をジョブコンテキストに格納する処理である。
  - ジョブコンテキストに格納したデータは以下の処理から参照することができる。
    - ジョブ前処理
    - ジョブ後処理
    - ビジネスロジック
    - Collector
    - 先頭チャンク前処理
    - 最終チャンク後処理

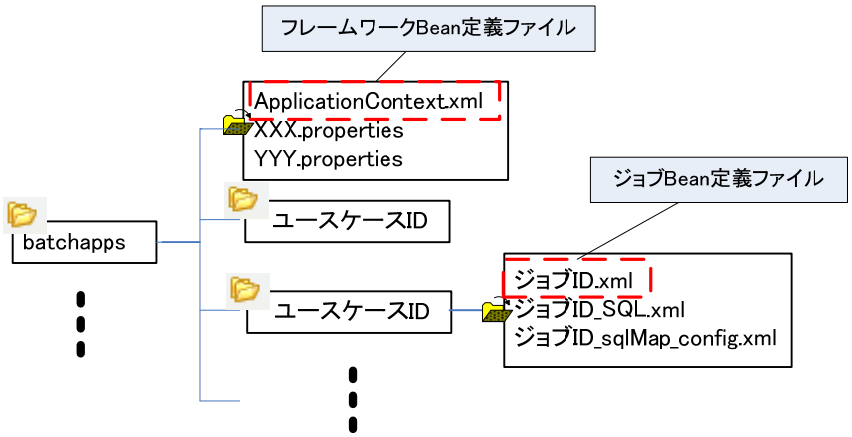
    詳細は『BA-01 トランザクション管理機能』、『BD-01 ビジネスロジック実行機能』、『BD-02 対象データ取得機能』を参照のこと。

- ジョブ終了コードの返却
  - ビジネスロジックの実装時、業務開発者が指定したジョブ終了コードをプロセスの終了コードとして返却することができる。
    詳細は『BD-01 ビジネスロジック実行機能』を参照のこと。
  - 指定されたジョブ終了コードは処理結果ハンドラに渡され返却される。処理結果ハンドラを独自に実装することでジョブ終了コード用処理を追加することができる。
    詳細は『BE-05 処理結果ハンドリング機能』を参照のこと。
  - ビジネスロジックからの指定が無い場合はフレームワークが規定するジョブ終了コードを返却する。詳細は『BE-05 処理結果ハンドリング機能』を参照のこと。

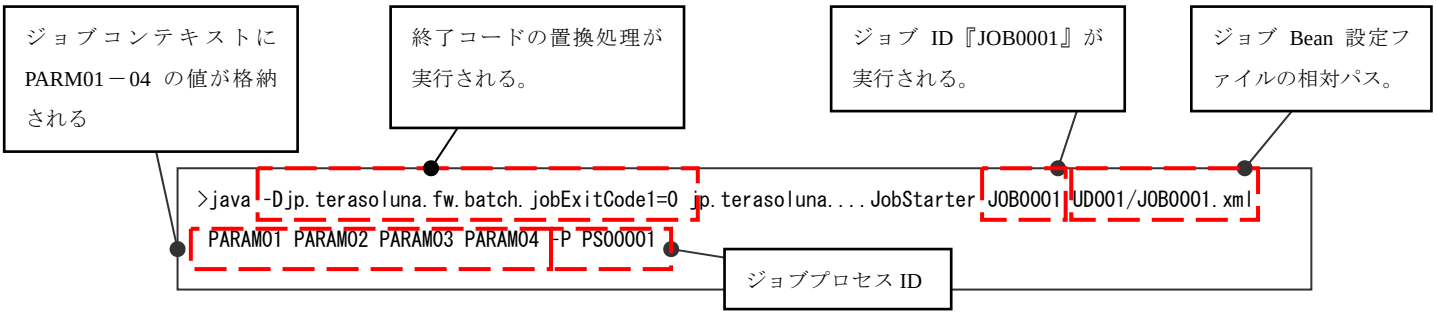
■ 使用方法

◆ コーディングポイント

- ファイル配置構成（例）



- フレームワークの起動例
  - クラスパスの設定
    - 「batchapps」フォルダをクラスパスに追加する。
  - 起動例

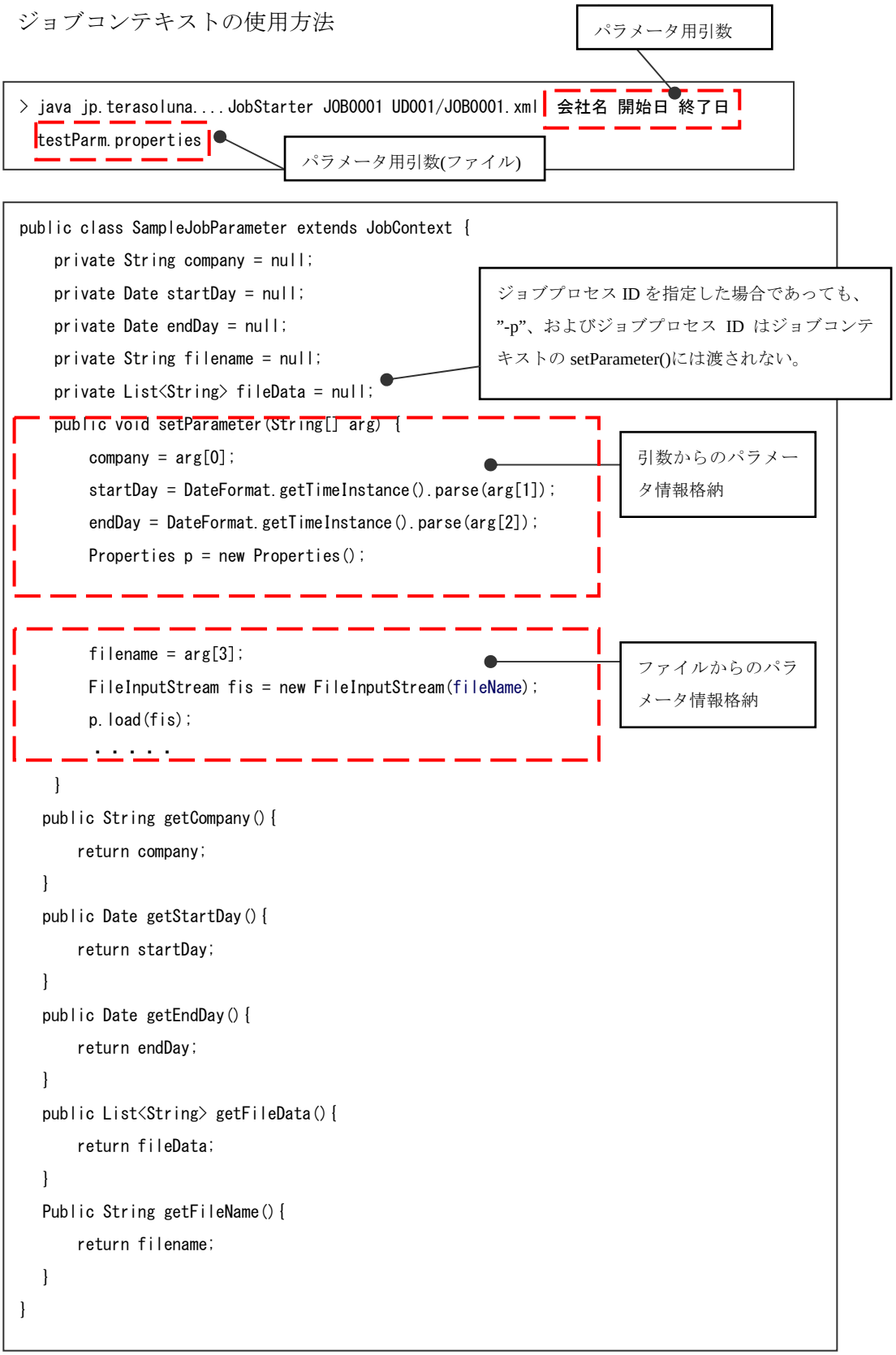


- ジョブ Bean 定義ファイルの設定例（ジョブ ID.xml）

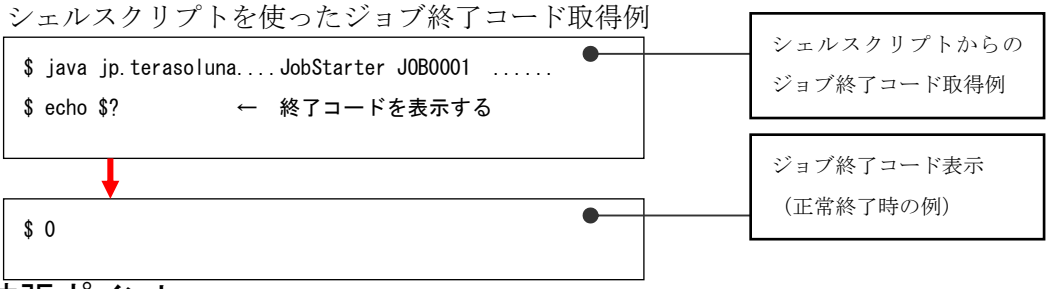
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
          "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
<!-- ジョブ実行処理用設定 -->
 <import resource="../../chunkTransactionBean.xml"/>
  .....
  .....
  </bean>
</beans>
```

チャンク別トランザクションモデルを使用した場合の例
詳細は『BA-01 トランザクション管理機能』を参照

- ジョブコンテキストの使用方法



- シェルスクリプトを使ったジョブ終了コード取得例



◆ 拡張ポイント

なし。

■ 関連機能

- 『BA-01 トランザクション管理機能』
- 『BD-01 ビジネスロジック実行機能』
- 『BD-02 対象データ取得機能』
- 『BE-03 ジョブ実行管理機能』
- 『BE-05 処理結果ハンドリング機能』

■ 使用例

- 機能網羅サンプル(functionsample-batch)
- チュートリアル(tutorial-batch)

■ 備考

- なし。

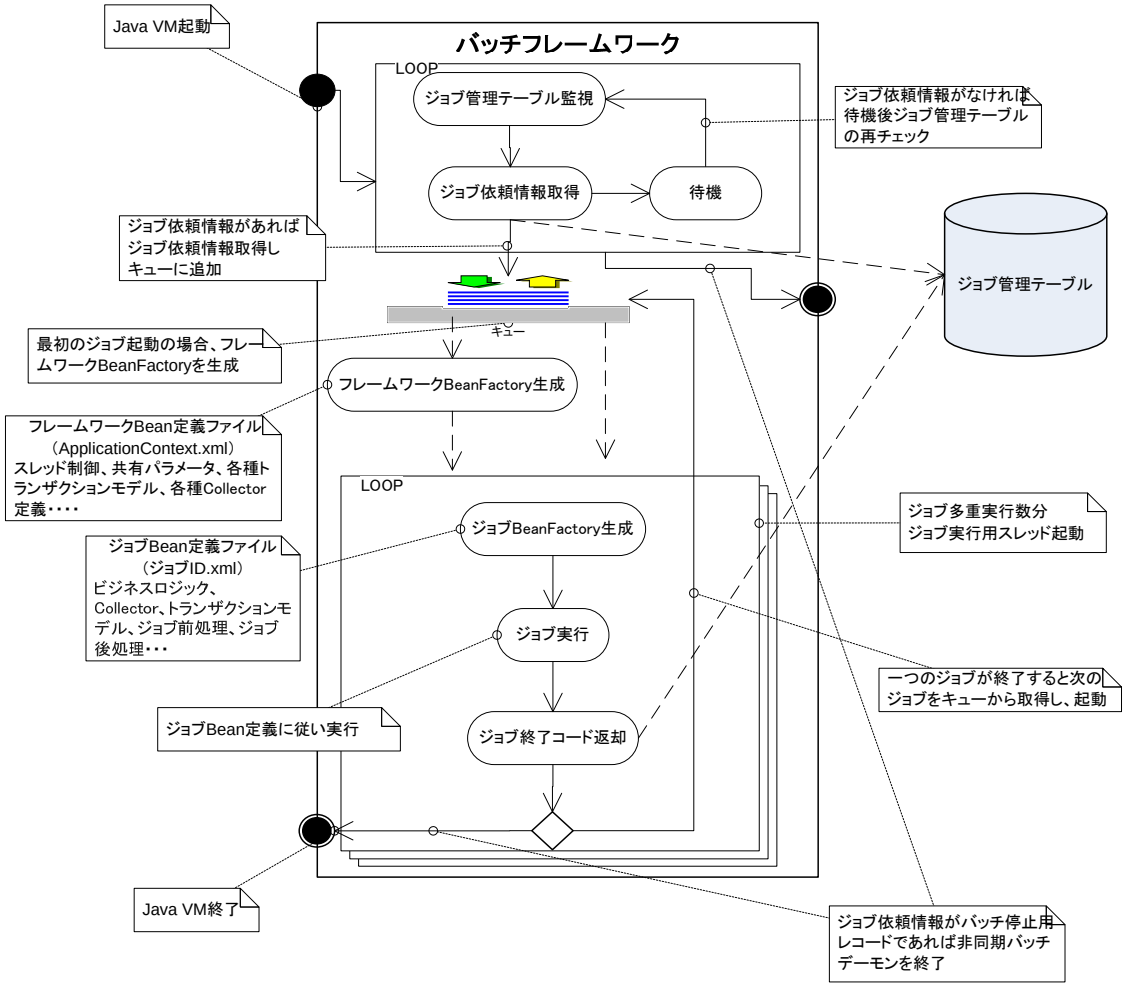
BE-02 非同期型ジョブ起動機能

■ 概要

◆ 機能概要

- フレームワークがジョブ管理テーブルを監視し、登録された対象のジョブを実行する機能である。
- 実行対象のジョブをジョブ管理テーブルに登録することで対象のジョブが実行される。
- ジョブ管理テーブルの監視周期、ジョブ多重実行数はデーモン用 Bean 定義ファイル(AsyncBatchDaemonBean.xml)から指定することがができる。

◆ 概念図



◆ 解説

- 非同期バッチデーモンの起動と終了
  (1) フレームワークは、デーモン起動用 Bean 定義ファイルに従い非同期バッチデーモンを起動する。
  (2) 非同期バッチデーモンは、フレームワーク Bean 定義ファイルを用いてフレームワーク BeanFactory の生成を行う。
    ✧ フレームワーク BeanFactory の生成処理に失敗した場合はエラーとして非同期バッチデーモンを終了する。
    ✧ 非同期バッチデーモンからジョブ多重実行を行う場合、スレッドプールが管理するスレッド数はジョブ多重実行数を考慮しフレームワーク Bean 定義ファイルに設定する必要がある。
  (3) 非同期バッチデーモンは、ジョブ管理テーブルを監視し、登録されたジョブがあればジョブ依頼情報を取得する。詳細は「ジョブ管理テーブルの監視、ジョブ依頼情報の取得、ジョブ起動」を参照のこと。
  (4) ジョブ依頼情報が取得された場合はジョブの起動状況をジョブ管理テーブルに登録する。詳細は「ジョブ管理テーブルの監視、ジョブ依頼情報の取得、ジョブ起動」を参照のこと。
  (5) ジョブ起動状況の登録が正常に終了した場合は対象ジョブを実行する。
    ① ジョブ Bean 定義ファイルを用いてジョブ BeanFactory の生成を行う。
    ② ジョブ BeanFactory の生成処理に失敗した場合はエラーとしてジョブを終了する。
    ③ ジョブ Bean 定義ファイルの設定内容に従いジョブを実行する。
  (6) ジョブの実行が終了するとジョブ終了コードをジョブ管理テーブルに登録する。
  (7) 取得したジョブ依頼情報がバッチ停止用レコードの場合は非同期バッチデーモンを終了する。
    ※ フレームワーク BeanFactory、ジョブ BeanFactory の詳細は『BE-01 同期型ジョブ起動機能』を参照のこと。

- ジョブ管理テーブルの監視、ジョブ依頼情報の取得、ジョブ起動
  - 指定された監視周期毎にジョブ管理テーブルをチェックし、ジョブ依頼情報を取得する処理である。
  - 非同期バッチデーモンが行うジョブ管理テーブル監視、ジョブ依頼情報取得の処理フローは以下である。
    ① 登録されたジョブ依頼情報があればジョブ依頼情報を取得する。
      - 一回の監視チェックで取得するジョブ依頼情報数は一つである。
    ② 取得したジョブ依頼情報に該当するレコードの「起動状況」を「起動中」に更新する。
    ③ ジョブ起動状況の登録に成功した場合のみジョブをキューに追加する。
      - 二重起動防止用の排他制御。
    ④ 対象のジョブ依頼情報がない場合は監視周期分待機した後、ジョブ管理テーブルのチェックを行う。
    ⑤ 非同期バッチデーモンが終了するまで上記の処理を継続する。
  - 非同期バッチデーモンのジョブ起動処理フローは以下である。
    ① 指定されたジョブ多重実行数分、ジョブ実行用スレッドが起動されキューからジョブを取得する。
      - 以下②～⑤の処理はジョブ実行用スレッド毎に実行される。
    ② キューから終了コードが取得された場合はジョブ実行用スレッドを終了する。
    ③ ジョブ実行用スレッドが終了するまで上記の処理を継続する。
  - ジョブ依頼情報の取得条件。
    - 「起動状況」が「起動前」であるもの。
    - 上記の条件かつ「ジョブ依頼番号」が一番小さいもの。
  - 非同期バッチデーモンの終了。
    - 取得したジョブ依頼情報がデーモン終了用ジョブ依頼であればキューに終了コードを追加し、ジョブ管理テーブル監視用スレッドを終了する。
    - デーモン終了用ジョブ依頼
      - ジョブ ID：「STOP」の文字例。（固定）
      - ジョブ Bean 定義ファイル名：「StopDaemonBean.xml」（変更可）

- ジョブ管理テーブル内容

項番	属性名	カラム名	必須	概要
1	ジョブ依頼番号	REQUEST_NO	○	ジョブ依頼連番。(DB の PK)
2	ジョブID	JOB_ID		実行対象のジョブ ID。(bean 名) 『STOP』を登録すると起動中のジョブ完了後、非同期バッチデーモンが終了される。
3	ジョブ Bean 定義ファイル名	JOB_FILE		実行対象のジョブ Bean が登録されているジョブ Bean 定義ファイルの相対パス。 デーモン終了用のジョブ ID 登録時は任意で設定すること。
4	パラメータ	PARAMETER		ジョブコンテキストに格納する値。 詳細は『BE-01 同期型ジョブ起動機能』を参照のこと
5	起動状況	STATE		0:起動前、1:起動中、3:正常終了、4:異常終了、7:中断/強制終了 バッチ登録時は 0 を初期値として登録する必要がある。
6	ジョブ終了コード	END_CODE		ジョブ処理終了後返却される終了コード。
7	更新時刻	UPDATE_TIME		更新時刻
8	登録時刻	REGISTER_TIME		登録時刻

※ ジョブ管理テーブルのテーブル名及びカラム名は各プロジェクトで自由に変更できる。

※ ジョブ Bean 定義ファイル名は別テーブル上で管理することも可能である。

※ ジョブ管理テーブルのジョブ依頼番号カラムの型はデフォルトでは可変長文字列となっている。対象ジョブ取得用 SQL を発行する際にソートを行うと辞書順にソートされることになる。

Ex) 登録順：
1,2,3,…,10,11,12,…20,21,22,…
取得順(ORDER BY REQUEST_NO)：
1,10,11,…,2,21,22,…

番号順にソートしたい場合は任意の桁に 0 パティングするなどの工夫が必要となる。

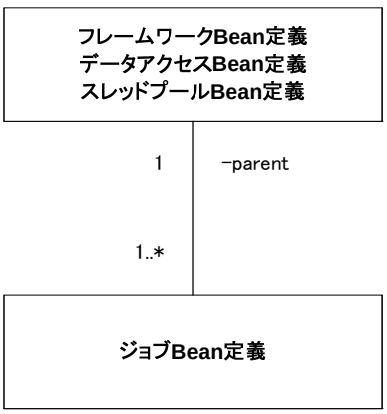
Ex) 登録例：
00001,00002,…,00010,00011,…,00020,00021,…
取得順(ORDER BY REQUEST_NO)：
00001,00002,…,00010,00011,…,00020,00021,…

- 非同期バッチデーモンの異常終了時対応
  予想外のエラーが発生し非同期バッチデーモンが終了するとジョブ管理テーブルの整合性の保証ができなくなる。(「起動状況」が「起動中」または「再起動中」のまま終了。)
  該当のジョブ依頼情報の「起動状況」を「起動前」に更新することでジョブを再開することができる。

- DB 障害時の運用
  ジョブ管理テーブルが存在する DB サーバがなんらかの障害で落ちた場合には、非同期バッチデーモンの再起動を行う必要がある。

- ジョブ BeanFactory のキャッシュ
  非同期バッチデーモンにジョブ BeanFactory をキャッシュさせることによって、ジョブ実行の度にジョブ BeanFacotry を作成しないようにし、ジョブ起動の性能向上を行うことができる。
  ジョブ BeanFactory をキャッシュする場合には、ジョブ Bean 定義ファイルに登録されている Bean のインスタンスが、そのまま次のジョブ起動時にも利用される。したがって、ジョブ Bean 定義ファイルに登録されている Bean のインスタンスが状態を持たないようにする必要がある。
  ジョブ BeanFactory のキャッシュを利用するかどうかは、ジョブ毎に指定することができる。ジョブ BeanFactory のキャッシュを利用する場合には、ジョブ Bean 定義ファイルに設定を行う。設定を行わなければ、キャッシュされない。

- フレームワーク BeanFactory とジョブ BeanFactory の生成
  同期型ジョブ起動と異なり、データアクセス Bean とスレッドプール Bean はフレームワーク BeanFactory として生成されフレームワーク全体でリソースを共有する。従って、生成するスレッド数はスレッドプール Bean 定義に設定する必要がありデーモン起動中は変更することができない。同じくデータアクセス Bean もフレームワーク BeanFactory として生成されるため一つのデーモンで設定できる SqlMapConfig ファイルは一つのみである。SqlMapConfig の詳細は『BA-02 データベースアクセス機能』を参照のこと。



- 複数の非同期バッチデーモンを立ち上げる場合について
  バッチデーモンを複数立ち上げて処理を行う場合、各処理を均等に振り分けることは出来ない。処理を均等に振り分けたい場合はバッチデーモンの拡張やジョブ管理テーブルを拡張して、ジョブ依頼情報の取得時にそれぞれのバッチデーモンが特定のジョブを所得するなどの工夫が必要となる。

- 複数の非同期バッチデーモンが立ち上がっている場合のデーモンの終了
  複数の非同期バッチデーモンが立ち上がっている場合に、ジョブ管理テーブルへバッチ停止用レコードを 1 件登録すると、デーモンは１つのみ終了される。この時終了するデーモンは、最初にバッチ停止用レコードを参照したデーモンとなるので、終了させたいデーモンはジョブ管理テーブルでは制御することができない。特定のデーモンを終了させたい場合は、デーモンのプロセスを直接停止する必要がある。
  全ての非同期バッチデーモンを終了させる場合は、デーモンが立ち上がっている数だけ、バッチ停止用レコードを登録する必要がある。

- 初期化処理エラー時の終了コードについて
  バッチデーモンから起動されるジョブが初期化処理などのエラーで異常終了する場合、同期型ジョブ起動では終了コード＝１を返却するが、非同期型ジョブ起動ではデフォルト終了コードに定義されている異常終了コードに変換されてジョブ管理テーブルに更新される。

- ジョブ個別の入力チェックについて
  フレームワーク BeanFactory とジョブ BeanFactory の生成でも述べているが、非同期ジョブ起動では、フレームワーク全体で幾つかのリソースを共有する。Validator も共有される一つであるため、各ジョブでの個別の入力チェックを実施する場合には、各ジョブ Bean 定義に個別の入力チェックに関する定義を別途設定する必要がある。
  - 非同期ジョブ起動時のジョブ個別入力チェックの定義例

```
<!-- コレクタの設定 -->
<bean id="collector"
class="jp.terasoluna.fw.batch.standard.StandardFileCollector">
  <property name="collectedDataHandlerFactory">
    <bean
      class="jp.terasoluna.fw.batch.validation.ValidationExecutorFactory">
      <property name="collectedDataHandlerFactory">
        <bean class="jp.terasoluna.fw.batch.standard.ChunkerFactory">
          <property name="chunkSize" ref="chunkSize" />
        </bean>
      </property>
      <property name="validator" ref="validator" />
      <property name="validationResultHandler" ref="validationResultHandler" />
    </bean>
  </property>
```

class を直接指定して、再定義。

```
  <property name="fileQueryDao" ref="csvFileQueryDAO" />
  <!-- 取得対象ファイル -->
  <property name="inputFileName" value="${base.dir}/inputfile/XX.csv"/>
  <property name="resultClass">
    <bean class="jp.co.nttdata.functiontest.common.bean.Member"/>
  </property>
  <property name="readNextLine" value="false" />
</bean>

<!-- 入力チェック(Validation)用ロケーション定義 -->
<util:list id="validationConfigLocations">
  <value>/common/validator-rules.xml</value>
  <value>/common/validator-rules-ex.xml</value>
   <value>/XX_validation.xml</value>
</util:list>

<!-- ハンドラ定義 -->
<bean id="validationResultHandler"
  class="jp.terasoluna.fw.batch.validation.Standar
  <!-- errorContinueFlag 設定：TRUE -->
  <property name="errorContinueFlg" value="true" />
  <property name="messageAccessor" ref="messageAccessor" />
</bean>

<!-- 入力チェック(Validation)Bean の定義 -->
<bean id="validator"
  class="org.springmodules.validation.commons.DefaultBeanValidator">
  <property name="validatorFactory">
    <bean parent="validatorFactory" />
  </property>
  <property name="useFullyQualifiedClassName" value="false" />
</bean>

<!-- 入力チェック(Validation)ルールの Factory の定義 -->
<bean id="validatorFactory"
class="jp.terasoluna.fw.validation.springmodules.DefaultValidatorFactoryEx">
  <property name="validationConfigLocations" ref="validationConfigLocations"
/>
</bean>
```

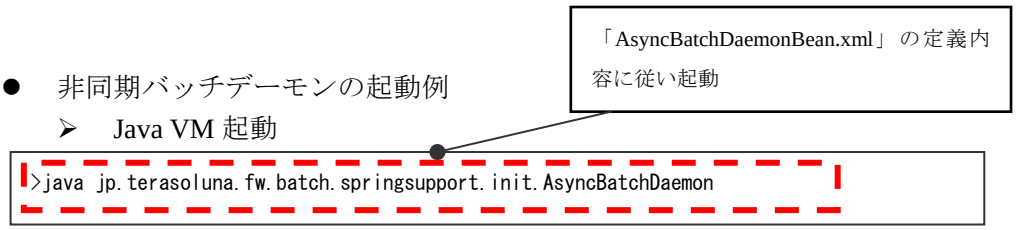
ジョブ個別の入力チェック定義

入力チェック関連の Bean の再定義
再定義をしない場合は、共有しているリソースを参照するため、個別の入力チェックが有効にならない。

■ 使用方法

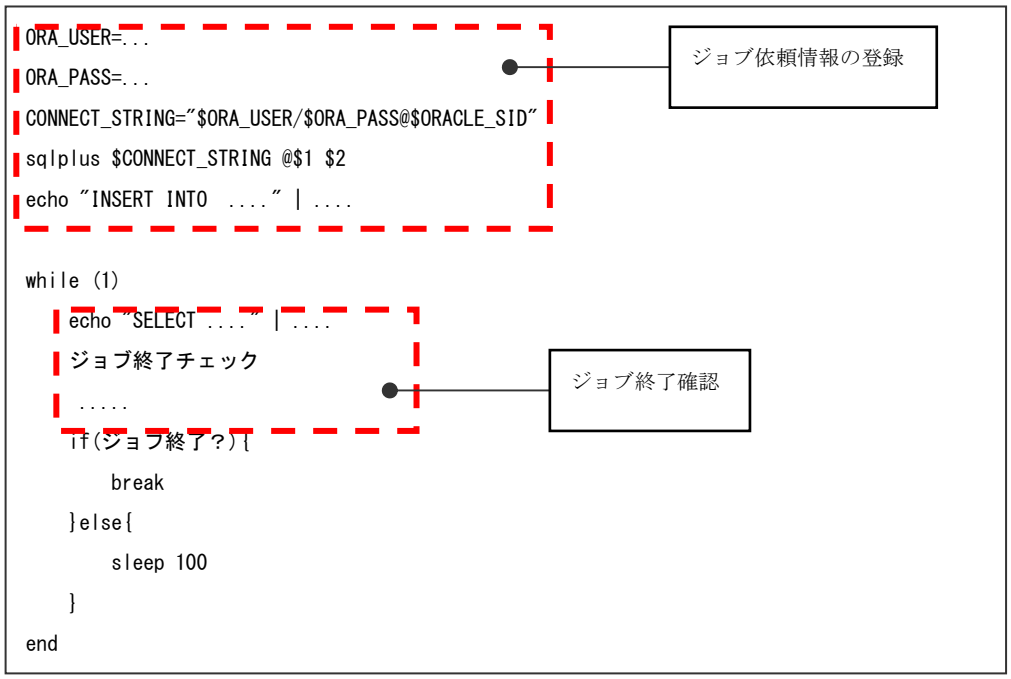
◆ コーディングポイント

- 非同期バッチデーモンの起動例
  - Java VM 起動

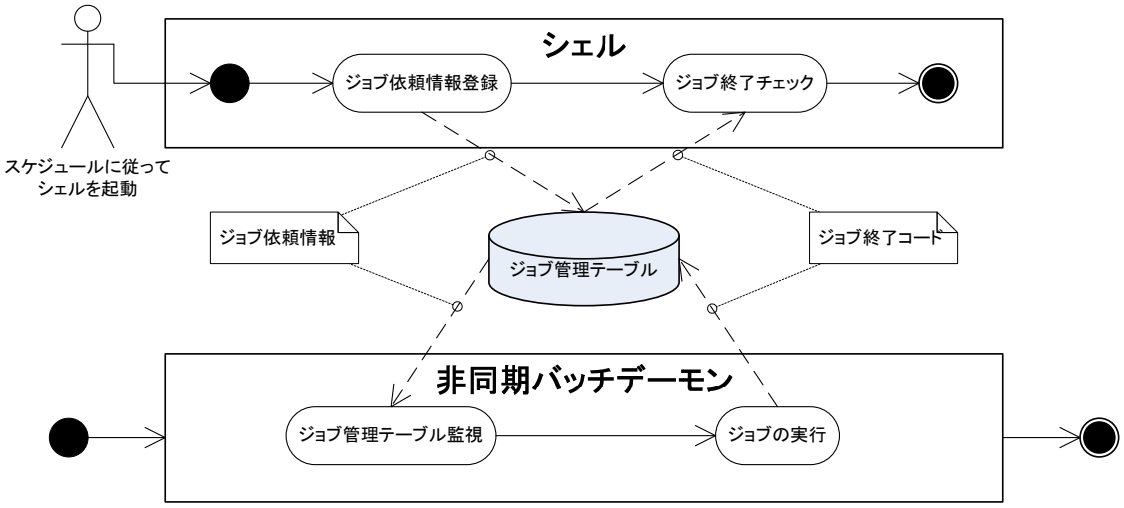


- スケジュールバッチの使用例
  スケジューラからスケジュールに従ってジョブ依頼情報をジョブ管理テーブルに登録することでスケジュールバッチに対応することが出来る。

  - ジョブ依頼情報登録用シェル



➢ スケジュールバッチの動作



- 周期バッチの使用例
  起動待ち解除時刻を利用し実行周期毎のジョブ依頼情報をジョブ管理テーブルへ大量に登録する方法や、スケジューラを利用し周期毎にジョブ依頼情報をジョブ管理テーブルへ登録する方法など、様々な方法から対応することができる。

- 随時バッチの使用例
  スケジューラやオンラインアプリケーションから必要に応じてジョブ依頼情報をジョブ管理テーブルに登録することでジョブを起動することができる。

- ジョブ BeanFactory のキャッシュ設定
  ジョブ BeanFactory をキャッシュする場合には、ジョブ Bean 定義ファイルに設定する。
  ➢ ジョブ BeanFactory をキャッシュする際のジョブ Bean 定義ファイルの設定

```
<!-- ジョブ BeanFactory のキャッシュ設定 -->
<util:constant id="useCache" static-field="java.lang.Boolean.TRUE" />
```

  ※ キャッシュする際に、同じジョブ Bean 定義ファイルを利用するジョブを複数起動した場合、ジョブごとに異なる必要がある Bean 定義(jobContext, JobStatus など)は、scope 属性を prototype にする必要がある。
  Bean 定義の scope 属性を singleton にすると全てのジョブで同じ Bean インスタンスを利用するためである。

◆ 拡張ポイント

- ジョブ管理テーブルの項目追加
  - 優先順位指定、起動待ち解除時刻等を追加し、ジョブの優先順位や起動時刻を管理することが出来る。
  - 拡張項目の起動条件の追加はジョブ依頼情報取得用 SQL 文を変更することで対応可能。
  - 拡張項目

項番	属性名	カラム名	必須	概要
1	優先順位	PRIORITY	○	優先順位。(小さい方が優先度が高い)
2	起動待ち解除時刻	WAIT_FOR_TIME		ジョブを起動する時刻。起動待ち解除時刻を経過したジョブ依頼情報のみ起動対象になる。 登録されたデータがない場合は経過したことにする。

  - ジョブ依頼情報の変更

拡張前の SQL 文

```
...
<select id="SELECTOO2" resultClass="……">
    SELECT
        REQUEST_NO    AS jobRequestNo ,
        JOB_ID        AS jobId ,
        JOB_FILE      AS jobDiscriptorPath ,
        PARAMETER     AS jobParameters,
        STATE         AS jobState,
        END_CODE      AS jobExitCode,
        UPDATE_TIME   AS updateTime,
        REGISTER_TIME AS registerTime
    FROM JOB_CONTROL
        WHERE REQUEST_NO = (
        SELECT
            MIN(REQUEST_NO)
        FROM
            JOB_CONTROL
        WHERE
            STATE = '0'
        )
 </select>
...
```

```
...
<select id="SELECT002" resultClass="……">
    SELECT
        REQUEST_NO    AS jobRequestNo ,
        JOB_ID        AS jobId ,
        JOB_FILE      AS jobDiscriptorPath ,
        PARAMETER     AS jobParameters,
        STATE         AS jobState,
        END_CODE      AS jobExitCode,
        UPDATE_TIME   AS updateTime,
        REGISTER_TIME AS registerTime
    FROM JOB_CONTROL
        WHERE REQUEST_NO = (
            SELECT
                MIN(REQUEST_NO)
            FROM
                JOB_CONTROL
            WHERE
                PRIORITY = (
                    SELECT MIN(PRIORITY)
                    FROM TEST001
                    WHERE
                        STATE = 0 AND
                        WAIT_FOR_TIME <= current_timestamp)
                )
</select>
…
```

拡張後の SQL 文

- 優先順位指定の使用例
  - ジョブ依頼情報の登録例

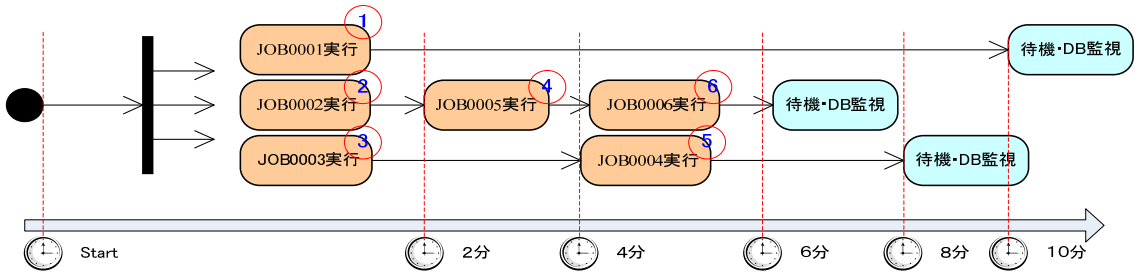
ジョブ依頼番号	ジョブID	ジョブ Bean 定義ファイル名	優先順位	起動状況	起動待ち解除時刻	更新時刻	登録時刻
0000000001	JOB0001	UC001/JOB0001.xml	1	0	2006/07/19 12:00:00	2006/07/19 12:00:00	2006/07/19 12:00:00
0000000002	JOB0002	UC001/JOB0002.xml	5	0	2006/07/19 12:00:00	2006/07/19 12:00:00	2006/07/19 12:00:00
0000000003	JOB0003	UC001/JOB0003.xml	5	0	2006/07/19 12:00:00	2006/07/19 12:00:00	2006/07/19 12:00:00
0000000004	JOB0004	UC001/JOB0004.xml	5	0	2006/07/19 12:11:00	2006/07/19 12:11:00	2006/07/19 12:11:00
0000000005	JOB0005	UC001/JOB0005.xml	1	0	2006/07/19 12:11:30	2006/07/19 12:11:30	2006/07/19 12:11:30
0000000006	JOB0006	UC001/JOB0006.xml	5	0	2006/07/19 12:00:13	2006/07/19 12:00:13	2006/07/19 12:00:13

※ 初期登録の不要属性は省略。

  - 上記ジョブ依頼情報例に基づく実行例
    - ジョブ起動開始時刻の条件 ： 2006/07/19 12:10:00
    - ジョブの実行時間の条件

ジョブ ID	処理時間	ジョブ ID	処理時間
JOB0001	10 分	JOB0002	2 分
JOB0003	4 分	JOB0004	4 分
JOB0005	2 分 10 秒	JOB0006	2 分

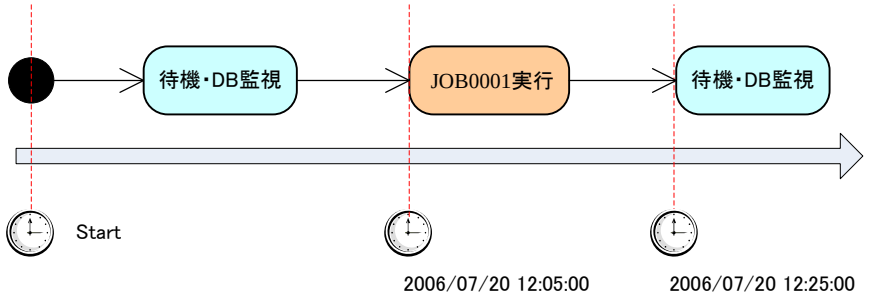
    - 上記の条件に従ったジョブの起動順



- 起動待ち解除時刻の使用例
  指定時刻以降に実行する必要があるジョブの実行管理ができる。
  例えば毎日の売り上げ状況を集計するジョブがあり、このジョブは毎朝５時以降に起動する必要がある場合「起動待ち解除時刻」の設定を行うことで複数のジョブを先に登録しておくことができる。「起動待ち解除時刻」は起動開始時刻ではないことに注意すること。
  - 実行例の条件
    監視周期：１０分
    起動スレッド数：１
    ジョブの処理時間：２０分
    ジョブ管理テーブルの登録例

ジョブ依頼番号	0000000001
ジョブID	JOB0001
ジョブ Bean 定義ファイル名	/UC0001/ JOB0001.xml
優先順位	1
起動状況	0
起動待ち解除時刻	2006/07/20 12:00:00
更新時刻	2006/07/19 12:00:00
登録時刻	2006/07/19 12:00:00

  - 動作
    「JOB0001」は「2006/07/20 12:00:00」から「2006/07/20 12:10:00」の間に起動される。
    ※ 指定時刻の定時に起動する必要があるジョブは同期バッチ起動方法を使う必要がある。詳細は『BE-01 同期型ジョブ起動機能』を参照のこと。



- デーモン終了用ジョブ依頼処理
  バッチデーモン終了用のジョブ依頼も通常のジョブと同じく起動し終了する。例えばジョブ ID が「STOP」である通常のジョブをジョブ管理テーブルに登録することで登録したジョブを最後にデーモンを終了させることが出来る。またその他デーモン終了時の処理を追加することができる。

■ 関連機能

- 『BE-01 同期型ジョブ起動機能』
- 『BA-02 データベースアクセス機能』

■ 使用例

- 機能網羅サンプル(functionsample-batch)
- チュートリアル(tutorial-batch)

■ 備考

- なし。

BE-03 ジョブ実行管理機能

■ 概要

◆ 機能概要

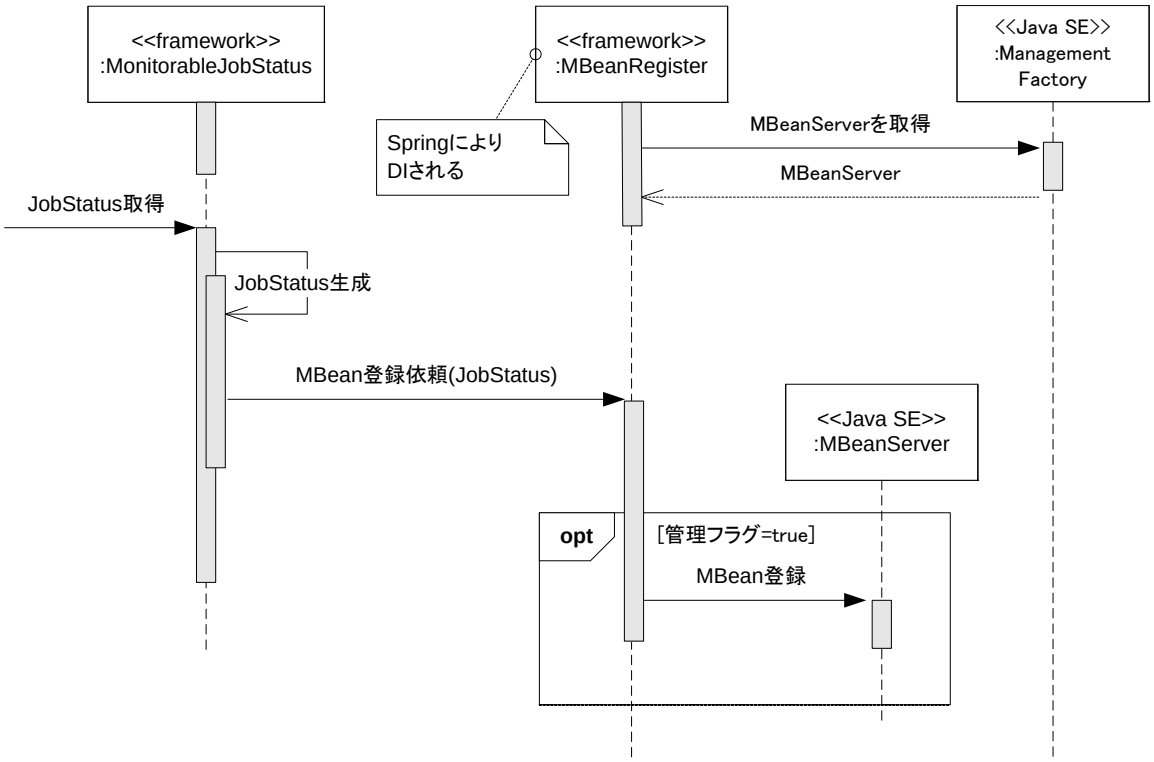
- JMX を使用したジョブ監視機能を提供する。
  - JMX とは Java Management Extensions テクノロジのことであり、Java Platform Standard Edition (Java SE) バージョン 5.0 の機能である。詳細は JDK 5.0 ドキュメントの『Java Management Extensions(JMX)概要』を参照のこと。

- 監視ツールは jconsole 等の JMX 用ツールを使用する。
  - jconsole は Java SE 5.0 に付属する JMX 準拠の監視ツールである。リモートホストの Java VM の状況監視が可能である。詳細は JDK 5.0 ドキュメントの『JDK ツールとユーティリティのドキュメント』を参照のこと。

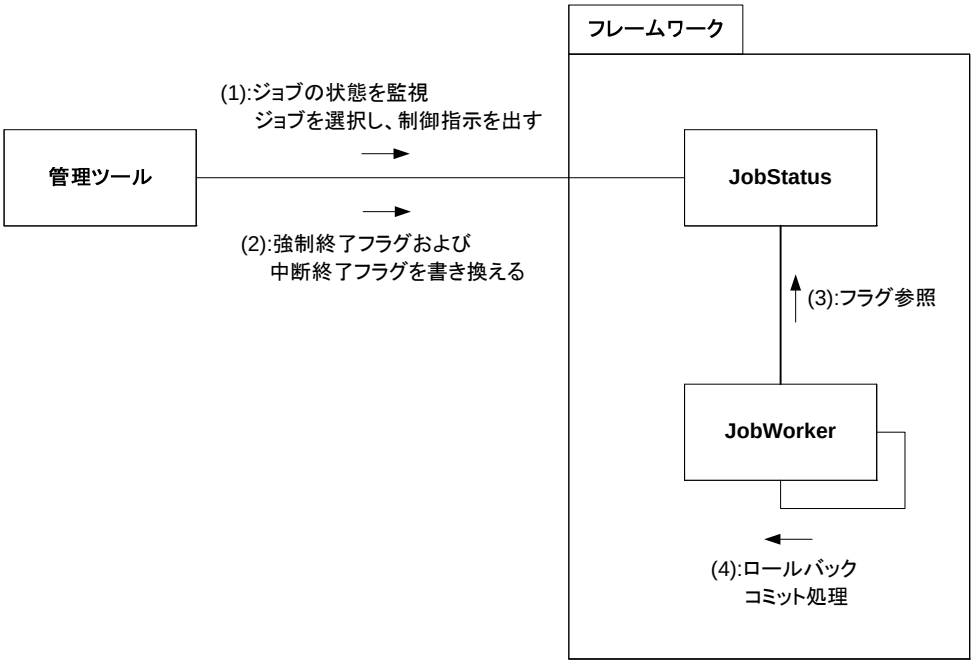
- ジョブ終了制御機能を提供する。
  - ジョブが実行中の場合のみ終了が可能。
  - 終了には強制終了と中断終了がある。強制終了はコミットされていないトランザクションをロールバックした後にジョブを終了する。中断終了は次のコミットポイントまで処理をし、コミット後に終了する。（詳細については、『**BE-03 補足資料:モデル別の中断・強制終了時の動作**』を参照のこと。）
  - 監視ツールから起動中のジョブを終了することができる。
  - 特定のディレクトリに終了ファイルを作成し、フレームワークがそのファイルを検出することでジョブを終了することができる。

◆ 概念図

- ジョブ監視機能の初期化処理について
  JMX では、管理対象のリソースを MBean として JMX サーバに登録する。本フレームワークでは MonitorableJobStatus を JMX サーバに登録する。
  MonitorableJobStatus は JobStatus を継承したクラスである。JMX による監視を行う場合には、ジョブ Bean 定義ファイルの useMonitorable を True に設定することで MonitorableJobStatus を利用するようにする。MonitorableJobStatus を利用することで、該当のジョブを監視することができる。ジョブ Bean 定義ファイルに useMonitorable の設定が無い場合はデフォルト Bean 定義ファイルの設定内容により監視有無が決まる。
  JMX サーバへの Mbean 登録処理は、MBean 登録クラスによって行われる。
  MBean 登録クラス（下図 MBeanRegister）は MonitorableJobStatus に DI されている。
  MBean 登録クラスが MBeanServer を取得し、MBean を MBeanServer に登録することで監視が行われる。MBean 登録名が既に MBeanServer に登録されている場合は、監視対象外とする。

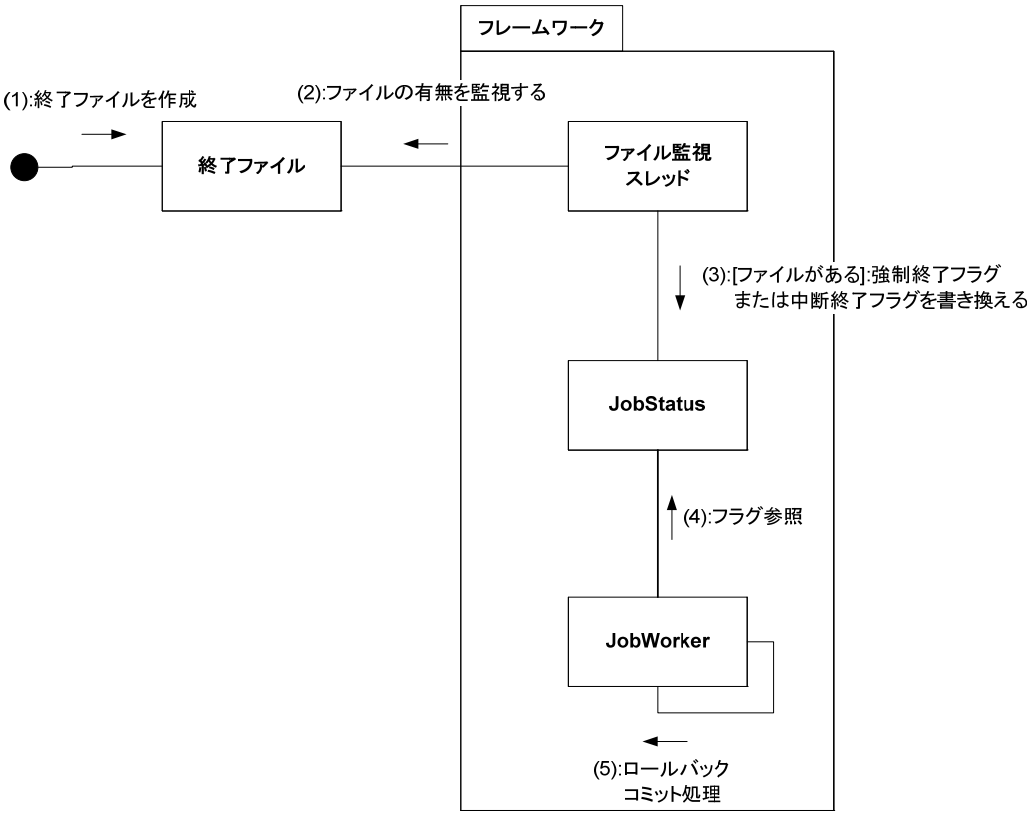


- 強制終了・中断終了について
  - 監視ツールを用いたジョブ強制終了および中断終了の流れ



(1) 監視ツールにて実行中のジョブを表示する。
(2) 監視ツールから選択された実行中のジョブに対して、強制終了フラグおよび中断終了フラグを書き換える。
(3) JobWorker が強制終了フラグおよび中断終了フラグを参照する。
(4) 強制終了の場合はロールバック処理を行う。中断終了の場合は次のコミットポイントまで処理を行った後、終了する。

➢ 終了ファイルを用いたジョブ強制終了および中断終了の流れ



(1) スケジューラまたはオペレータ操作から終了ファイルを作成する。フレームワークで規定する終了ファイル検出ディレクトリに強制終了または中断終了ファイルが作成される。

項番	項目名	フォーマットの形式	具体例
1	同期型強制 終了ファイル	[ジョブ ID]_[ジョブプロセス ID].end [ジョブ ID].end（ジョブプロセス ID 設定無）	JOB0001_0001.end JOB0001.end
2	同期型中断 終了ファイル	[ジョブ ID]_[ジョブプロセス ID].irp [ジョブ ID].irp（ジョブプロセス ID 設定無）	JOB0001_0001.irp JOB0001.irp
3	非同期型強制 終了ファイル	[ジョブ ID]_[ジョブ依頼番号].end	JOB0001_001.end
4	非同期型中断 終了ファイル	[ジョブ ID]_[ジョブ依頼番号].irp	JOB0001_001.irp

※ジョブプロセス ID 付きジョブの制御は[ジョブ ID]または[ジョブ ID]_[ジョブプロセス ID]の型式のみ有効である。

ジョブプロセス ID とは、同期型ジョブ起動での起動において同じジョブ ID のジョブを起動する際に区別するための識別子である。オプションとして起動時に設定する。同期型ジョブ起動および非同期型ジョブ起動については

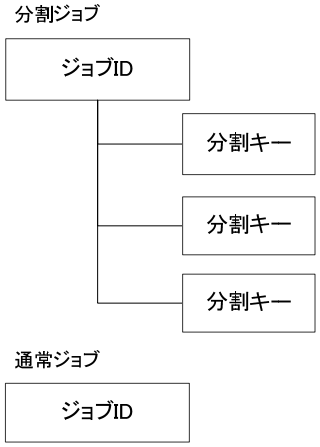
『BE-01 同期型ジョブ起動機能』、『BE-02 非同期型ジョブ起動機能』を参照。

(2) 指定した間隔毎に終了ファイル検出ディレクトリをチェックする。
(3) (2)で終了ファイルが検出されると、終了ファイルの種類を判断し、JobStatus の強制終了フラグおよび中断終了フラグを書き換える。
(4) JobWorker が強制終了フラグおよび中断終了フラグを参照する。
(5) 強制終了の場合はロールバック処理を行う。中断終了の場合は次のコミットポイントまで処理をし、コミットする。なお、作成された終了ファイルはフレームワークでは削除しない。

◆ 解説

- ジョブ管理設定
  起動するジョブ毎にジョブ管理の有無を設定することができる。ジョブ管理を「無」にした場合、該当のジョブは監視ツールによる監視およびジョブ終了制御が不可となる。ただし、終了ファイルを用いたジョブ終了制御は可能である。

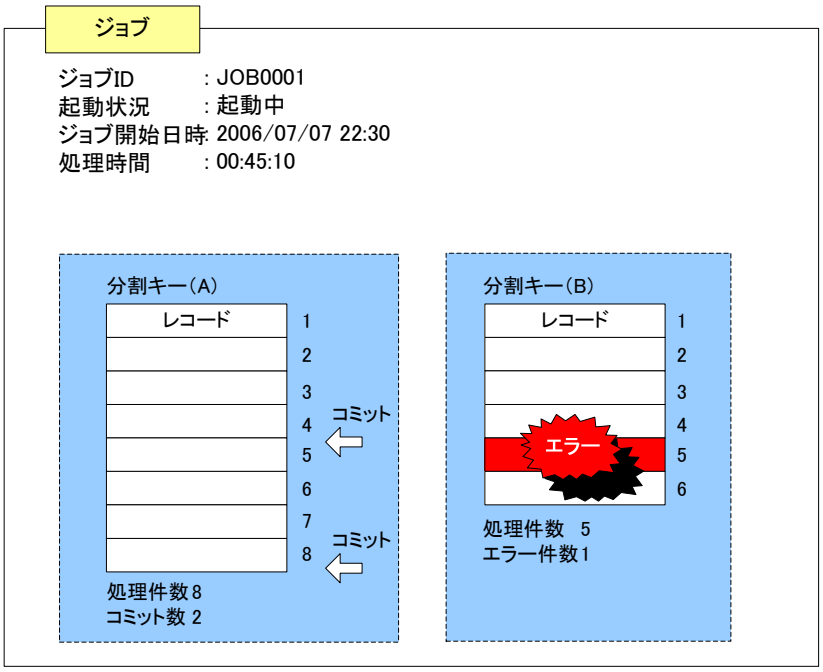
- 監視ツールでの表示方法
  監視ツールでは、監視内容を階層化して監視ツールに表示する。
  同期型ジョブ起動で起動されたジョブの場合には、表示の際の表示名はジョブ ID とジョブプロセス ID（ジョブプロセス ID の指定が無い場合はジョブ ID のみ）である。非同期型ジョブ起動で起動されたジョブの場合には、表示名はジョブ ID とジョブ依頼番号である。



- 監視属性について
  監視属性は JobStatus に定義し、JMX 仕様に準拠した MBean インタフェースを実装している。インタフェースには監視属性に対する getter メソッドが定義される。
  - 監視属性

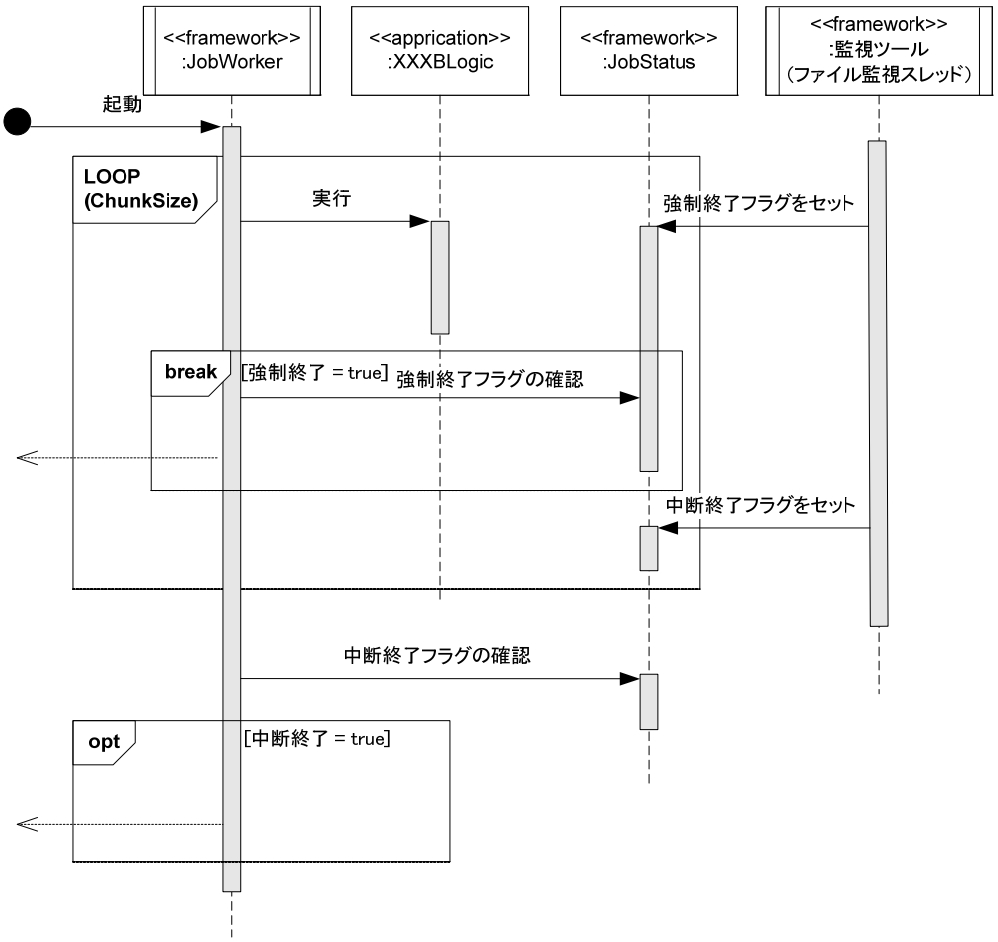
項番	項目名	概要
1	ジョブ ID	ジョブ毎に設定する ID
2	起動状況	実行したジョブの状況を表す 起動中、再開起動中、正常終了、異常終了、強制終了、中断終了
3	ジョブ終了コード	ジョブ終了後に返却される終了コード
4	ジョブ開始日時	ジョブを開始した日時
5	処理時間	ジョブ開始日時から算出した実行時間
6	分割キー	ジョブ分割の場合は分割キーを表示 通常ジョブの場合はなし
7	コミット数	コミットした回数
8	処理件数	リターンコードで NOMAL_CONTINUE が設定された BLogicResult の総数
9	エラー件数	リターンコードで ERROR_CONTINUE が設定された BLogicResult の総数

起動状況については『BE-02 非同期型ジョブ起動機能』参照のこと。



- 監視ジョブ数
  JobStatus がインスタンス化され、MBeanServer に登録された時点で監視が可能となる。1 つの Java VM 内で監視できるジョブ数の上限を設定することができ、上限値を超えると処理が終了したジョブから監視が解除される。

- 強制終了および中断終了時の JobWorker の動作
  強制終了の場合は、JobStatus に強制終了フラグがセットされると JobStatus が JobWorker に対して強制終了の指示を出し、ジョブを終了する。
  中断終了の場合は、1 トランザクション毎にコミットした後に中断終了フラグを確認する。中断終了の指示があればジョブを終了する。



- 強制終了および中断終了でのジョブの終了について
  同期型ジョブ起動ではジョブの終了時に Java VM も終了する。一方、非同期型ジョブ起動では対象のジョブのみ終了する。

- 分割ジョブの終了について
  監視ツールを使用する場合は、分割キー毎にジョブ終了制御が可能となる。終了ファイルを用いたジョブ終了制御の場合は、分割キー毎に終了することはできない。

- 強制終了ができない場合について
  強制終了は、フレームワークがスレッドに対する割り込みを検知することで実現されている。したがって、以下のような場合には強制終了によってジョブを終了させることができない。
  - アプリケーションのメソッドが実行を続けている場合
    （無限ループを実行しているなど）
  - アプリケーションでデッドロックが発生し、実行が止まっている場合

■ 使用方法

◆ コーディングポイント

- JMX エージェントの起動
  リモートシステムからの監視とジョブ終了制御を可能にするには、Java VM 起動時に、システムプロパティを設定する必要がある。詳細は『JDK ツールとユーティリティのドキュメント』参照のこと。

項番	プロパティ	内容
1	com.sun.management.jmxremote.port	JMX/RMI 接続を有効にしたいポート番号を指定する
2	com.sun.management.jmxremote.ssl	SSL の有効/無効を設定する
3	com.sun.management.jmxremote.authenticate	パスワード認証の有効/無効を設定する

設定例

```
>java -Dcom.sun.management.jmxremote.port=9999
-Dcom.sun.management.jmxremote.ssl=false
-Dcom.sun.management.jmxremote.authenticate=false
```

Java VM 毎に変更する

- デフォルト Bean 定義ファイルの設定方法
  - ジョブ監視の有無に関する設定例

```
<util:constant  id="useMonitorable"
     static-field="java.lang.Boolean.FALSE"/>
</util:constant>
```

ジョブ監視の有無

  - ジョブ管理上限値の設定例

```
<bean id="manageableJobSize" class="java.lang.Integer">
        <constructor-arg value="20" />
</bean>
```

監視ツールで管理可能なジョブの上限値

  - ジョブ終了制御に関する設定例

```
<bean id="endFileChecker"
  class="jp.terasoluna.fw.batch.init.EndFileChecker">
    <property name="endFileDir" value="batchapps/BE-03/EndFile" />
</bean>
<bean id="scheduledTask"
  class="org.springframework.scheduling.timer.ScheduledTimerTask">
    <property name="period" value="8000"/>
    <property name="timerTask" ref="endFileChecker"/>
</bean>
```

終了ファイル検出ディレクトリ

チェック間隔(ミリ秒で指定)

- 終了ファイルを用いた強制終了および中断終了の注意点
  終了ファイル検出ディレクトリのチェック間隔は、処理するデータの件数と性能を考慮して適切な値を設定すること。チェック間隔が短い場合、パフォーマンスが低下する。

- 強制終了および中断終了を行うための指針
  フレームワークでは終了ファイルを作成する機能はなく、終了ファイルを置くタイミングはフレームワーク側では関与しない。以下に終了ファイルを置くタイミングと手段の指針を示す。
  - タイミング
    ジョブが予定時間内に終了しない場合や、メンテナンスなどの理由により実行中のバッチプログラムを終了させたい場合等。
  - 手段
    オペレータや停止用ジョブの実行により終了ファイルを配置する。

◆ 拡張ポイント

- なし。

■ 関連機能

- 『BE-01 同期型ジョブ起動機能』
- 『BE-02 非同期型ジョブ起動機能』

■ 使用例

- 機能網羅サンプル(functionsample-batch)
- チュートリアル(tutorial-batch)

■ 備考

- なし。

補足資料: モデル別の中断･強制終了時の動作

■ 概要

中断終了と強制終了では、トランザクション処理の動作が異なる。
また、トランザクションモデル毎に中断・強制終了時のトランザクションおよびジョブ終了のタイミングが異なる。

◆ **中断終了**
次のコミットポイントまで処理をし、コミット後にジョブを終了する。
ただし、『非トランザクションモデル』ではトランザクション管理を行わないため、中断指示後も処理が継続して実行される。

◆ **強制終了**
コミットされていないトランザクションをロールバックし、ジョブを終了する。

◆ **トランザクションモデル**
フレームワークで以下の３つを提供している。詳細については『BA-01 トランザクション管理機能』を参照のこと。

1. チャンク別トランザクションモデル
   ✧ チャンク単位に、トランザクションで処理する。

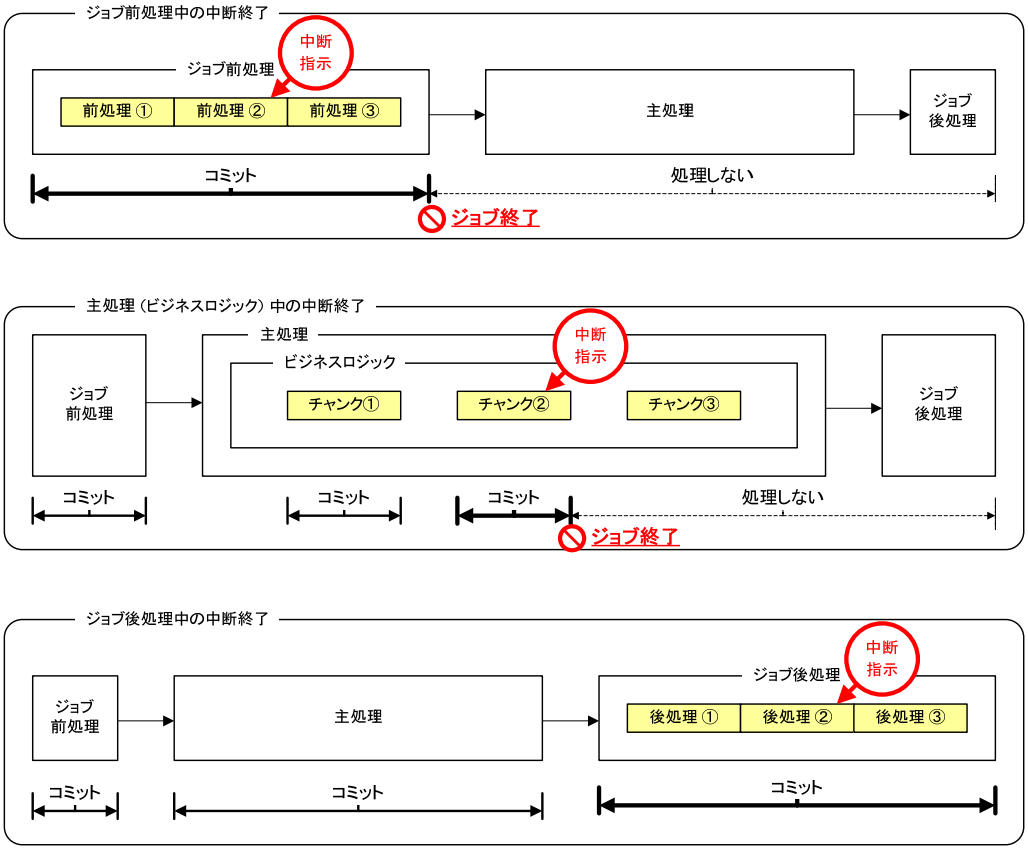
2. 全チャンク単一トランザクションモデル
   ✧ すべてのチャンクを単一のトランザクションで処理する。

3. 非トランザクションモデル
   ✧ トランザクション管理を行わない。

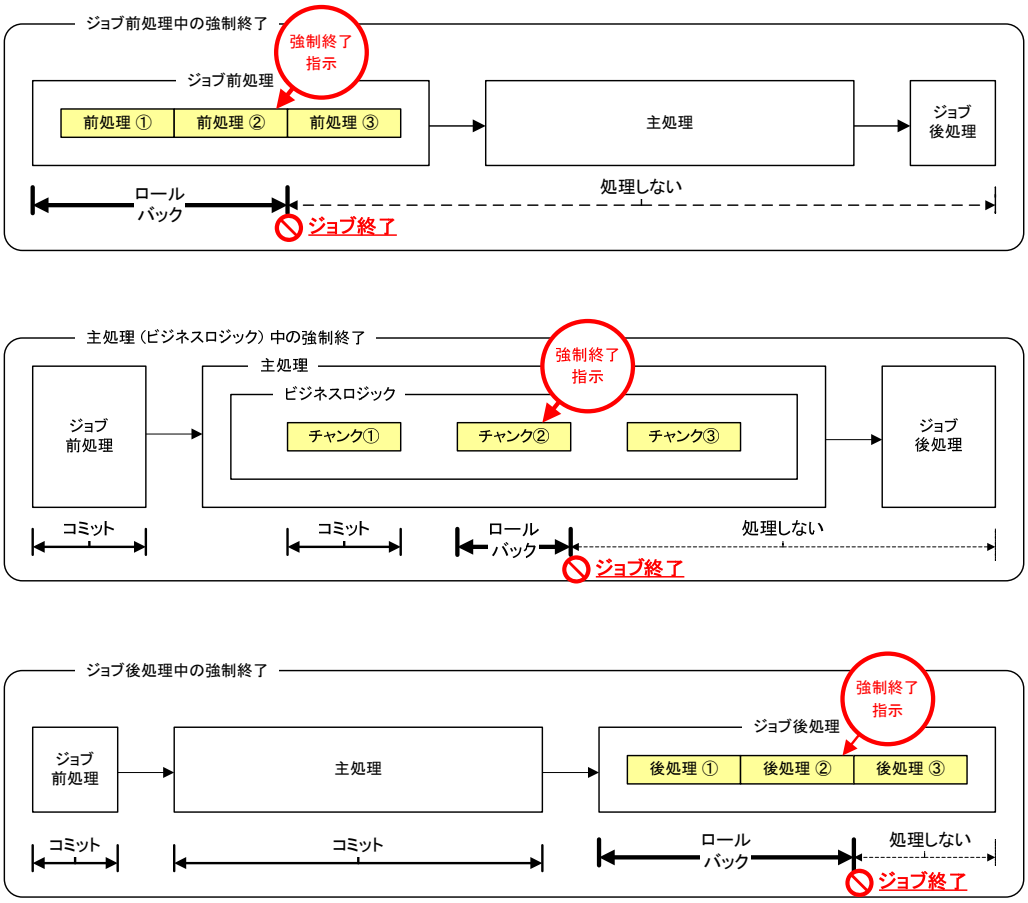
①『チャンク別トランザクションモデル』の中断・強制終了

ジョブ前処理／主処理／ジョブ後処理の動作は下図のとおりである。
中断・強制終了した場合は、全て **SUSPEND** のジョブ終了コードが返却される。

【中断終了時の動作】



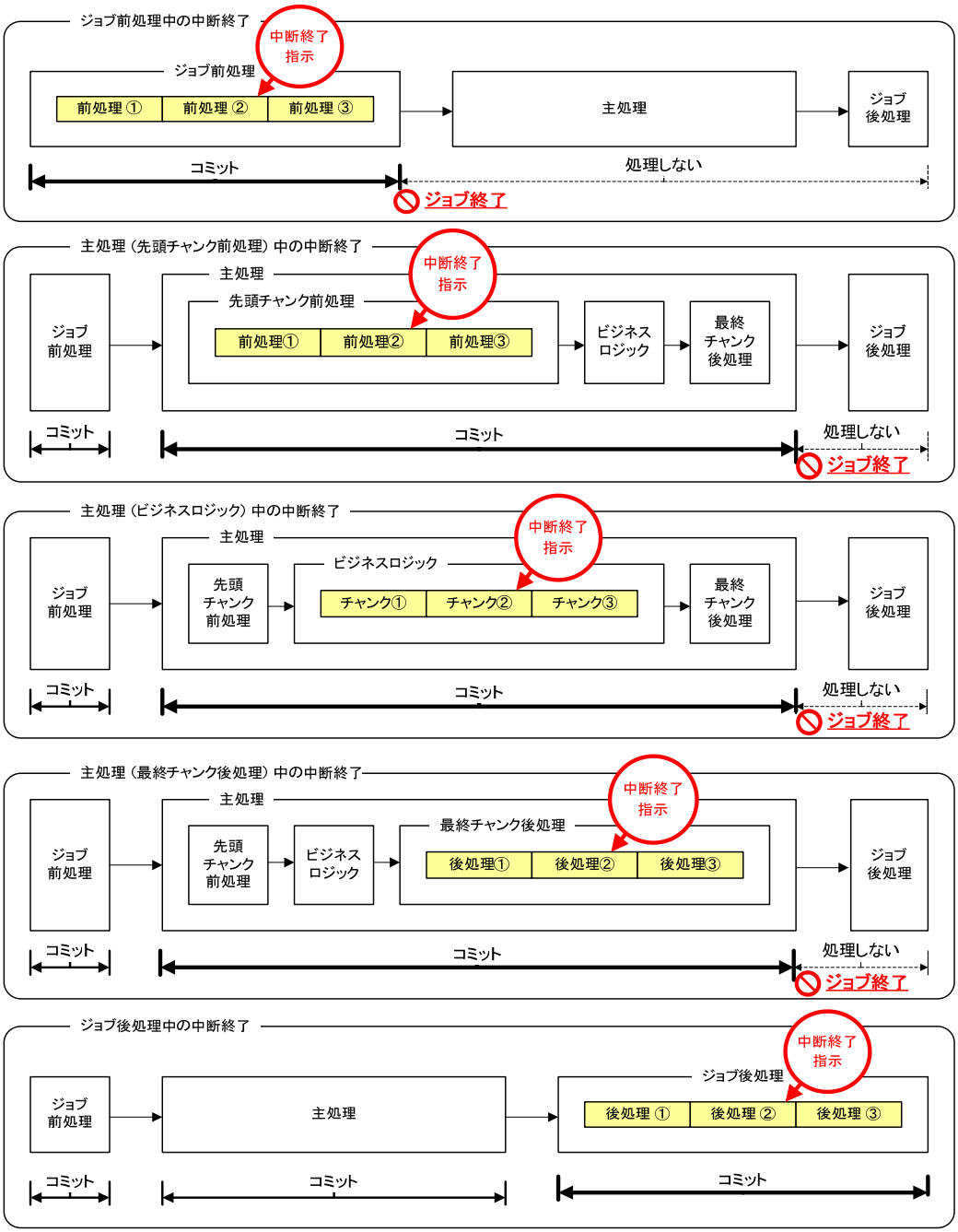
【強制終了時の動作】



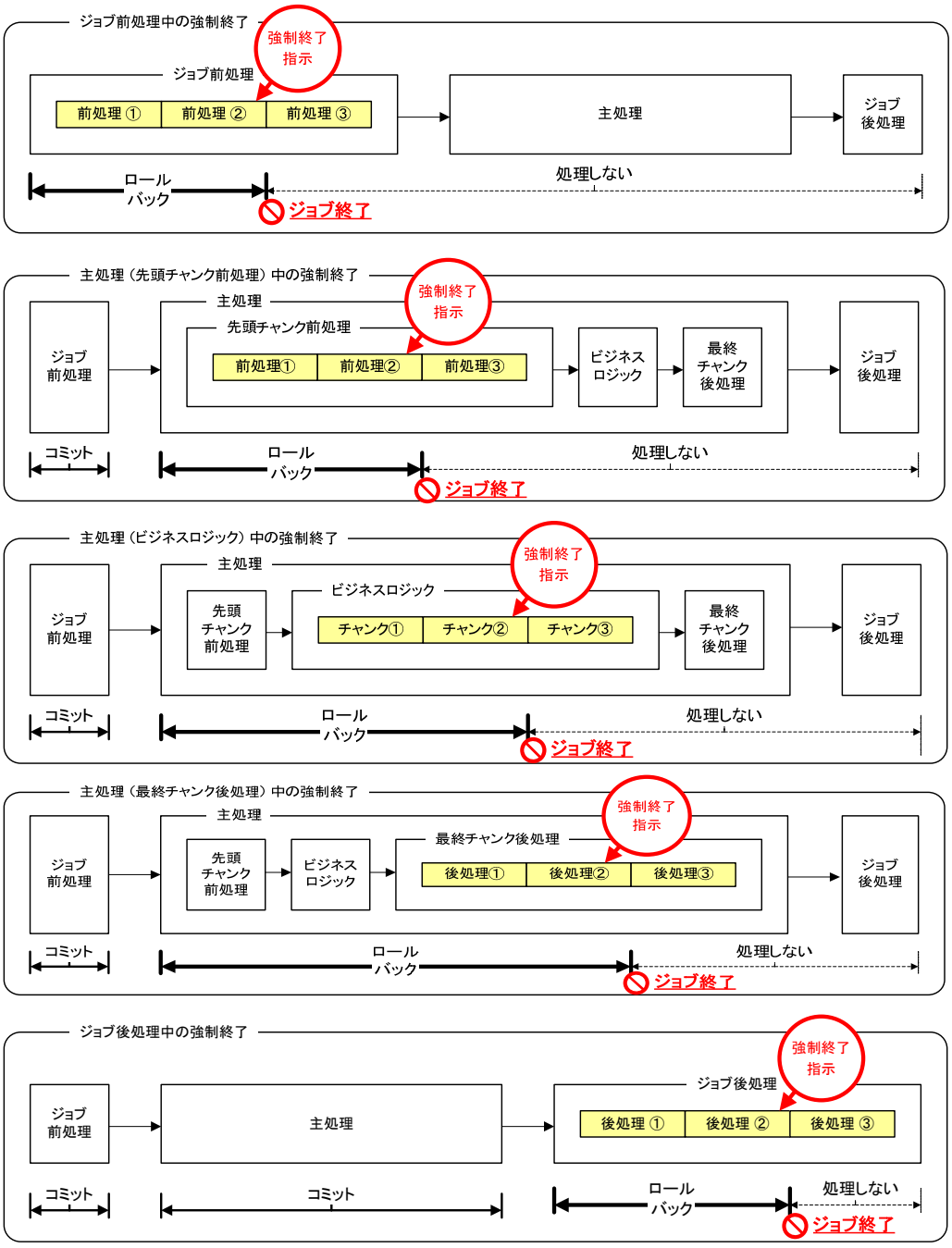
②『全チャンク単一トランザクションモデル』の中断・強制終了

ジョブ前処理／主処理／ジョブ後処理の動作は下図のとおりである。
中断・強制終了した場合は、全て **SUSPEND** のジョブ終了コードが返却される。

【中断終了時の動作】



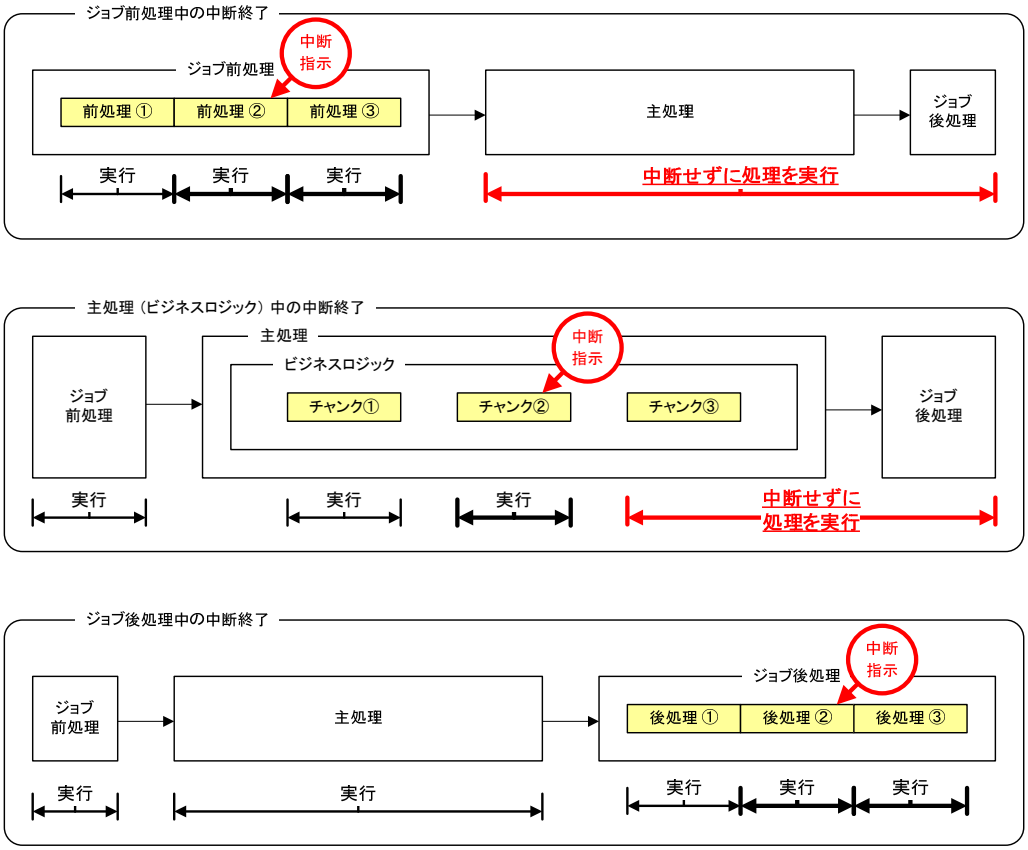
【強制終了時の動作】



③『非トランザクションモデル』の中断・強制終了

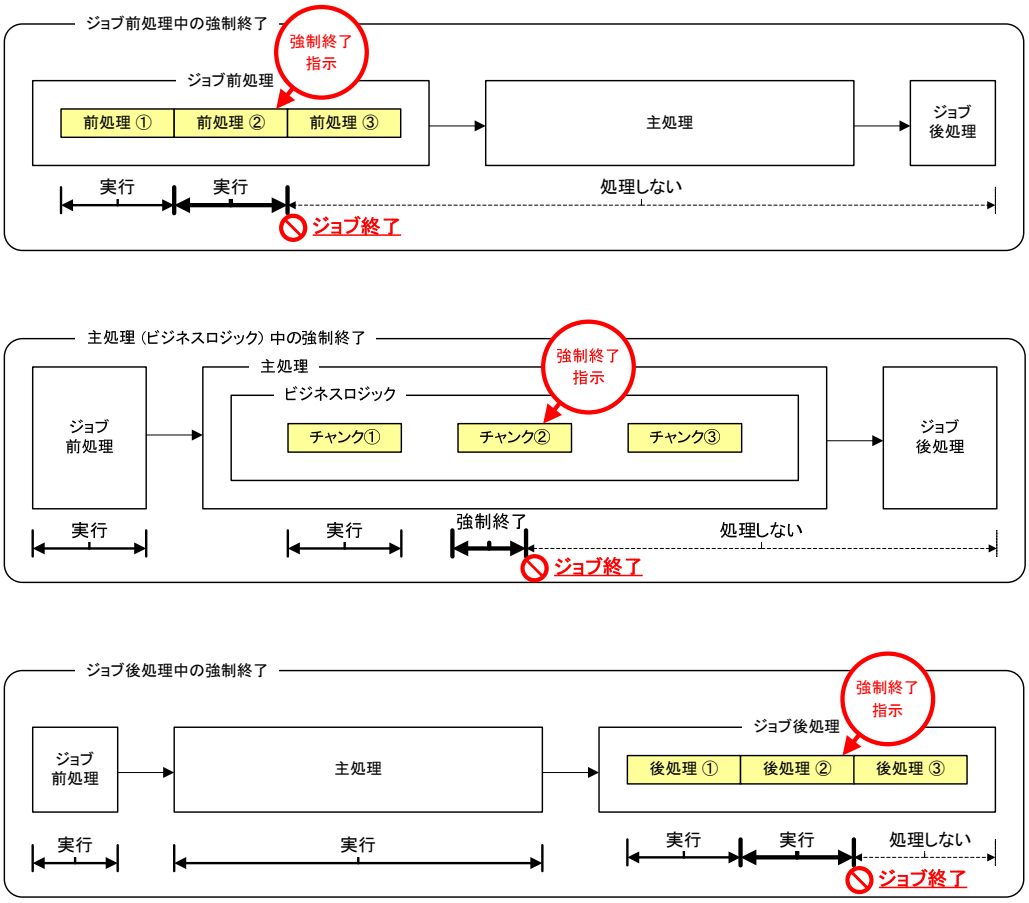
ジョブ前処理／主処理／ジョブ後処理の動作は下図のとおりである。
中断・強制終了した場合は、全て **SUSPEND** のジョブ終了コードが返却される。

【中断終了時の動作】



※『非トランザクションモデル』ではトランザクション管理を行わないため、中断指示後も処理が継続して実行される。

【強制終了時の動作】



■ 関連機能

- 『BA-01 トランザクション管理機能』
- 『BE-03 ジョブ実行管理機能』

■ 使用例

- なし。

■ 備考

- なし。

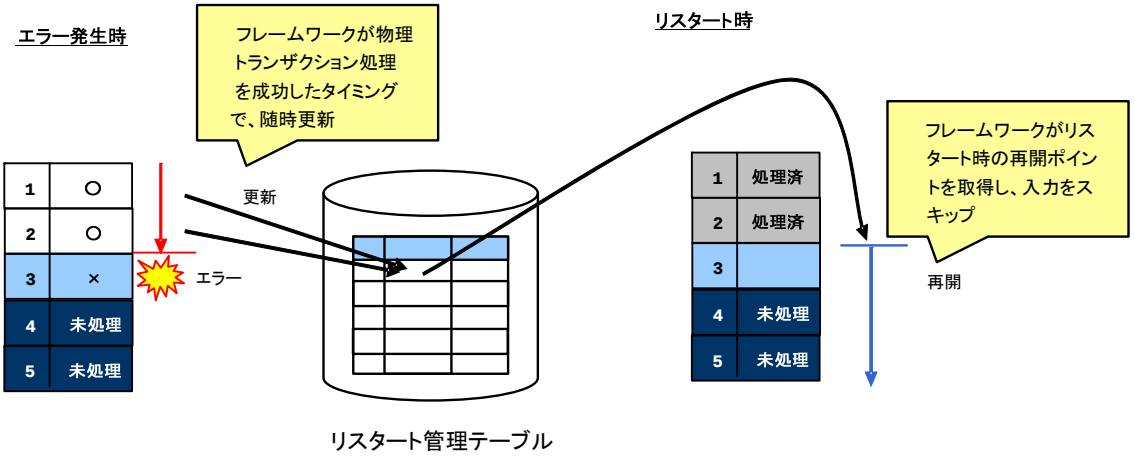
BE-04 リスタート機能

■ 概要

◆ 機能概要

- 中断されたジョブの再開時、リスタート管理テーブルに記録されたリスタートポイント以降のデータから処理をリスタートする機能を提供する。
  分割ジョブの場合は分割したキー毎に処理をリスタートすることが出来る。
- フレームワークは以下の処理を提供する。
  - リスタートポイント、ジョブコンテキストの登録・更新処理。
  - リスタート時のジョブコンテキストの復元処理。
  - リスタート時のリスタートポイント以降の対象データ取得処理。
  - リスタート情報のクリア処理。

◆ 概念図



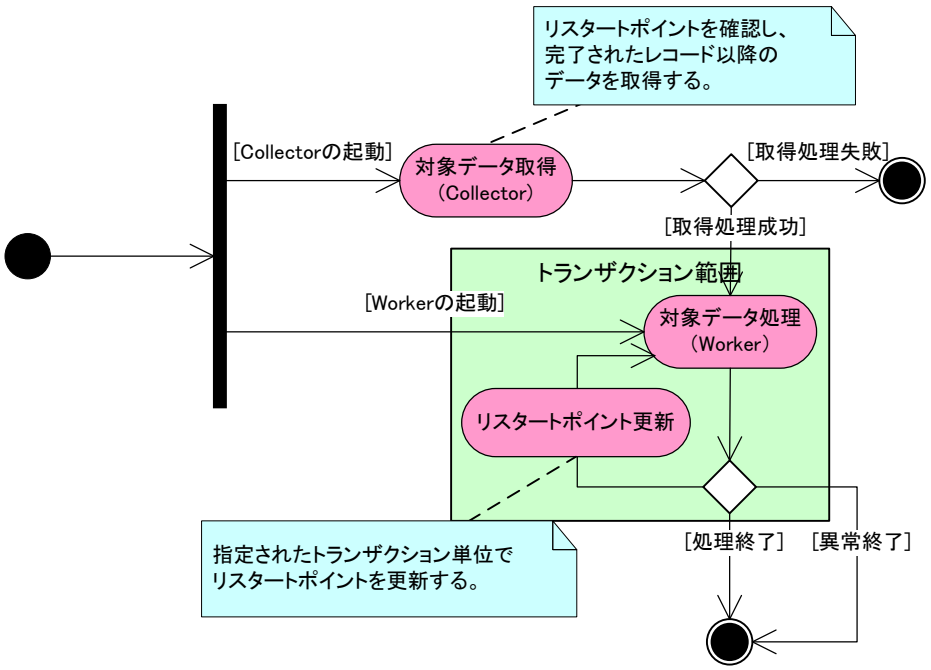
◆ 解説

- リスタート処理の起動条件
  - 以下の Bean 定義ファイル雛形をジョブ Bean 定義ファイルに import することでリスタート機能を使用することができる。
    - 「ChunkTransactionForRestartBean.xml」：リスタート機能有効チャンク別トランザクションモデル用の雛形
    - 「PartitionChunkTransactionForRestartBean.xml」：リスタート機能有効、分割ジョブ、チャンク別トランザクションモデル用の雛形
      （詳細は『BA-01 トランザクション管理機能』を参照すること。）
  - フレームワークはリスタート管理テーブルのリスタート情報を参照し起動方法を判断する。

項番	リスタート情報	処理状況	起動内容
1	登録なし	－	通常のジョブ起動
2	登録あり	処理中	リスタート起動
3	登録あり	処理完了	起動されない

  - リスタート情報はジョブ単位で登録されるが、対象のジョブが分割ジョブである場合はジョブ全体と分割キー毎のリスタート情報がそれぞれ登録される。

- リスタートポイント、ジョブコンテキストの登録・更新処理
  - リスタート機能を有効に設定することでフレームワークはリスタート管理テーブルにリスタートポイント、ジョブコンテキストを登録・更新する。
  - リスタートポイント、ジョブコンテキストが初期登録されるのはビジネスロジック実行時の最初のトランザクションである。ジョブ前処理を実行している段階では、リスタートポイント、ジョブコンテキストは登録されない。
  - リスタートポイント、ジョブコンテキストの更新処理はトランザクション毎に行う。トランザクションの詳細は『BA-01 トランザクション管理機能』を参照すること。
  - リスタートポイント、ジョブコンテキストの更新処理はビジネスロジックのトランザクションと同一トランザクションであるためエラーが発生してもデータ不整合は発生しない。



- リスタート時のジョブコンテキストの復元
  - フレームワークは初期化処理時にリスタート管理テーブルからジョブコンテキストを取得し、ジョブコンテキスト状態を処理中断時の状態に復元する。
  - リスタート時にはジョブ前処理は実行されない。ただし、前回実行時にジョブ前処理でジョブコンテキストにセットされたデータはジョブコンテキストが復元されるため、リスタート時にも利用することができる。
  - 分割ジョブでは、ジョブコンテキストの復元処理はリスタート管理テーブルのリスタート情報に基づき親ジョブ、子ジョブそれぞれについて行われる。
    - ✧ 親ジョブのジョブコンテキストは、親ジョブの初期化処理時に復元される。
    - ✧ 子ジョブのジョブコンテキストは、子ジョブの初期化処理時に復元される。
    - ✧ 親ジョブのジョブコンテキスト復元処理に失敗した場合は、ジョブをエラー終了する。
    - ✧ 子ジョブのジョブコンテキスト復元処理に失敗した場合は、対象の子ジョブのみエラー終了する。

- リスタート情報のクリア処理
  - リスタート機能が有効であり、ジョブ処理が正常に完了した場合はリスタート情報のクリア処理を行う。
    - ✧ リスタート管理テーブルの該当データを削除する。
  - 分割ジョブでは、リスタート情報のクリア処理は親ジョブ、子ジョブそれぞれ行われる。
    - ✧ リスタート情報のクリア処理のトランザクション

項番	ジョブの種類	ジョブ後処理定義がある場合	ジョブ後処理定義がない場合
1	通常のジョブ	定義されたジョブ後処理と同一トランザクション	ジョブ後処理として行われる
2	分割ジョブ（親ジョブ）	定義されたジョブ後処理と同一トランザクション	ジョブ後処理として行われる
3	分割ジョブ（子ジョブ）	分割された子ジョブ毎の子ジョブ後処理と同一トランザクション	分割されたキー毎のジョブ後処理として行われる

ジョブ後処理の詳細は『BD-01 ビジネスロジック実行機能』を参照のこと。

- ✧ 親ジョブ用のリスタート情報クリア処理に失敗した場合はジョブをエラー終了する。
- ✧ 子ジョブ用のリスタート情報クリア処理に失敗した場合は対象の子ジョブをエラー終了する。この場合、親ジョブは他の子ジョブ全体が終了してからエラー終了する。

- リスタート管理テーブル

項番	項目名称	カラム名	概要
1	ジョブリクエスト番号	REQUEST_NO	非同期型ジョブ起動時はジョブ依頼番号、同期型ジョブ起動時はジョブプロセス ID である。
2	ジョブ ID	JOB_ID	リスタート対象のジョブ ID。
3	分割キー値	PARTITION_KEY	リスタート対象のジョブ分割キー値。 分割ジョブの親ジョブ、および分割ジョブではないジョブの場合はジョブ ID が登録される。
4	リスタートポイント	RESTART_POINT	処理を完了(コミット)したレコード数。
5	ジョブコンテキスト	JOB_CONTEXT	ジョブコンテキストをシリアライズ化したデータ。
6	処理状況	STATE	1:処理中、2:処理完了 「処理中」のみリスタート対象になる。

※ テーブルの主キーは「ジョブリクエスト番号」、「ジョブ ID」、「分割キー値」である。

※ リスタート管理テーブルのテーブル名及びカラム名は各プロジェクトで自由に変更できる。

※ リスタート管理テーブルはビジネスロジックと同じトランザクション上で扱えるテーブルである必要がある。

- 分割ジョブのリスタート
  - 分割キー毎にリスタートを管理することが出来る。
  - 分割ジョブが中断されリスタートする場合は子ジョブ全体がリスタート対象になる。子ジョブ毎のリスタートを選択する必要がある場合はリスタート管理テーブルの「処理状況」の変更やリスタート情報を削除することで通常のジョブ起動、リスタート起動、起動なしを選択することができる。

- リスタート時の対象データ取得

  ジョブ対象データを取得する際、リスタートポイントを確認し、完了されたレコード以降のデータのみ取得する機能を提供する。詳細は『BD-02 対象データ取得機能』を参照のこと。

  起動毎に対象データの取得結果が変わるようなジョブはリスタートポイントから完了されたレコードを判別することができないためリスタート機能を利用することが出来ない。

- リスタート機能使用時の注意事項
  - 同期型ジョブ起動時にリスタート機能を利用する場合は引数でジョブプロセス ID を指定することを推奨する。ジョブプロセス ID の指定がない場合は、ジョブ ID のみを識別子としてリスタート情報が参照される。そのため、一度あるジョブ ID のリスタート情報が登録されると、同一ジョブ ID のジョブは起動されなくなる。同期型ジョブ起動時の引数に関する詳細は『BE-01 同期型ジョブ起動機能』を参照のこと。
  - ジョブの起動がリスタートになる場合は起動時に指定したパラメータは無効になり、リスタート時にパラメータを指定することはできない。但し、TERASOLUNA-Batch ではジョブコンテキスト復元処理を提供し前回実行時指定されたパラメータ値は復元される。
  - リスタート情報が登録されるのはビジネスロジック実行時の最初のトランザクションであり、最初のトランザクションが失敗するとリスタート情報は登録されない。従って前処理が存在するジョブが前処理段階でエラー終了した場合のリスタートは、リスタート情報が登録されていないため、ジョブ前処理が再度実行される。
  - フレームワークでは、ファイル出力等のトランザクショナルでないリソースに対しては、トランザクション処理は行わないためファイル出力を行うジョブのリスタートは正しく動作しない。ファイル等の非トランザクションリソースの扱いの詳細は『BA-01 トランザクション管理機能』を参照のこと。
  - フレームワークが提供するリスタート機能は ERROR_CONTINUE で処理されたデータのリスタートは対象外とする。従って ERROR_CONTINUE で処理されたデータのリスタートはビジネスロジック実装又は運用で対応する必要がある。ERROR_CONTINUE の詳細は『BD-01 ビジネスロジック実行機能』を参照のこと。
  - リスタート情報として登録されるものは、「対象データの何件目のデータまで処理が成功したか」を示す件数だけである。したがって、実行された時点で対象データの件数や並び順が変更されるジョブでは、フレームワークで提供するリスタート機能を用いてリスタートすることはできない。

■ 使用方法

◆ コーディングポイント

リスタート機能を使用する場合は、
「ChunkTransactionForRestartBean.xml」
「 PartitionChunkTransactionForRestartBean.xml 」
のどちらかを設定する。

- ジョブ Bean 定義ファイルの設定例

```
...
<!-- ジョブ固有の設定情報：チャンク別トランザクション  -->
<import resource="../../ChunkTransactionForRestartBean.xml"/>
...
```

- ジョブプロセス ID の使用例
  - 同期型ジョブ起動時にジョブプロセス ID を指定することで、起動タイミングが異なる同一ジョブ ID のジョブを識別することができる。ジョブプロセス ID を識別子として指定することで、起動タイミングが異なる同一ジョブ ID のジョブ毎にリスタートすることができる。
  - ジョブプロセス ID には、ジョブの起動日付等のジョブの起動を一意に識別できるものを設定する。

◆ 拡張ポイント

- ジョブコンテキストのサイズが 4000 バイト以上になる場合の対応(Oracle の場合)

説明：機能網羅サンプルなどで定義している RAW 型だと 4000byte までしかバイナリを格納できない。これを回避する方法として BLOB 型を使用する設定に変更を行う。

前提条件：Oracle10g の JDBC ドライバを使用すること。
Oracle9.2 以前の JDBC ドライバを使用した場合は、入力の途中で、予想外のファイルの終了、または予想外のストリームの終了があったことを表す `java.io.EOFException` が発生する。

1. Restart_Control テーブル定義を変更する。
   RAW 型から BLOB 型に変更する

No	ﾌｨｰﾙﾄﾞ名	型	長さ	NOT NULL	ﾃﾞﾌｫﾙﾄ値
1	REQUEST_NO	VARCHAR2	20	NOT NULL	
2	JOB_ID	VARCHAR2	20	NOT NULL	
3	PARTITION_NO	VARCHAR2	20	NOT NULL	
4	PARTITION_KEY	VARCHAR2	30		
5	RESTART_POINT	NUMBER	30,0		
6	**JOB_CONTEXT**	**RAW**	**2000**		
7	STATE	CHAR	1		
8	UPDATE_TIME	TIMESTAMP(6)			
9	REGISTER_TIME	TIMESTAMP(6)			

図 1　変更前

No	ﾌｨｰﾙﾄﾞ名	型	長さ	NOT NULL	ﾃﾞﾌｫﾙﾄ値
1	REQUEST_NO	VARCHAR2	20	NOT NULL	
2	JOB_ID	VARCHAR2	20	NOT NULL	
3	PARTITION_NO	VARCHAR2	20	NOT NULL	
4	PARTITION_KEY	VARCHAR2	30		
5	RESTART_POINT	NUMBER	30,0		
6	JOB_CONTEXT	BLOB			
7	STATE	CHAR	1		
8	UPDATE_TIME	TIMESTAMP(6)			
9	REGISTER_TIME	TIMESTAMP(6)			

図 2 変更後

2. sqlMapClient が BLOB を扱えるように設定を変更する。
[dataAccessContext-batch.xml]を編集する。

```
<!-- iBATIS データベース層のための SQlMap の設定 -->
<bean id="sqlMapClient"
      class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="configLocation" ref="sqlMapConfigFileName" />
  <!—- LOB Handler の設定 -->
  <property name="lobHandler" ref="oracleLobHandler"/>
</bean>

<!-- LOB Handler で使用する JdbcExtractor 設定-->
<bean id="simpleExtractor"
 class="org.springframework.jdbc.support.nativejdbc.SimpleNativeJdbcExtractor"/>
<!—- LOB Handler の設定 -->
<bean id="oracleLobHandler"
 class="org.springframework.jdbc.support.lob.OracleLobHandler">
<property name="nativeJdbcExtractor" ref="simpleExtractor"/>
</bean>
```

BLOB を使用するためのハンドラを DI

JDBCExtactor を定義

Oracle 用の Lob ハンドラを定義

3. iBatis の BLOB を扱えるように設定を追加する。
リスタート管理テーブル用 sqlMap を設定している sqlMapConfilg に設定を追加する。
[common/sqlMapConfig.xml] を編集

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMapConfig PUBLIC "-//iBATIS.com//DTD SQL Map Config 2.0//EN"
 "http://www.ibatis.com/dtd/sql-map-config-2.dtd">
<sqlMapConfig>

  <settings useStatementNamespaces="true" />

  <!—- BLOB を Byte 配列で扱う Handler の設定 -->
  <typeHandler
    callback="org.springframework.orm.ibatis.support.BlobByteArrayTypeHandler"
        javaType="[B"
        jdbcType="BLOB"/>

  <sqlMap resource="common/jobControl-sqlMap.xml"/>
  <sqlMap resource="common/jobRestart-sqlMap.xml"/>
  <sqlMap resource="common/jobResult-sqlMap.xml"/>

</sqlMapConfig>
```

BLOB を Byte 配列で扱うための設定

4. SELECT 文の取得結果を BLOB でマッピングするように指定する。
[common/jobRestart-sqlMap.xml] を編集する。

```
<!-- リスタート依頼情報取得 -->
<resultMap class="jobRestartInfo" id="jobRestartInfoMap">
    <result property="requestNo" column="requestNo"/>
    <result property="jobId" column="jobId"/>
    <result property="partitionKey" column="partitionKey"/>
    <result property="restartPoint" column="restartPoint"/>
    <result property="jobContext" column="jobContext" jdbcType="BLOB"/>
    <result property="state" column="state"/>
    <result property="updateTime" column="updateTime"/>
    <result property="registerTime" column="registerTime"/>
</resultMap>

<select id="SELECT_JOB_RESTART_INFO"
        parameterClass="jobRestartInfo"
 resultMap="jobRestartInfoMap">
   SELECT REQUEST_NO    AS requestNo,
          JOB_ID       AS jobId,
          PARTITION_KEY AS partitionKey,
          RESTART_POINT AS restartPoint,
          JOB_CONTEXT   AS jobContext,
          STATE        AS state,
          UPDATE_TIME   AS updateTime,
          UPDATE_TIME AS registerTime
     FROM RESTART_CONTROL
    WHERE REQUEST_NO   = #requestNo#
      AND JOB_ID      = #jobId#
      AND PARTITION_NO = #partitionNo#
</select>
```

リスタート依頼情報クラスと SQL の結果をマッピングする
ジョブコンテキストは BLOB を明示的にする

5. INSERT,UPDATE 文のパラメータを BLOB でマッピングするように指定する。
[common/jobRestart-sqlMap.xml] を編集する。

```
<!-- リスタート情報更新 -->
<update id="UPDATE_JOB_RESTART_POINT" parameterClass="jobRestartInfo">
  UPDATE RESTART_CONTROL
     SET RESTART_POINT = #restartPoint# ,
         JOB_CONTEXT   = #jobContext:BLOB#,
         STATE         = #state#,
         UPDATE_TIME   = current_timestamp
    WHERE REQUEST_NO   = #requestNo#
      AND JOB_ID      = #jobId#
      AND PARTITION_NO = #partitionNo#
</update>

<!-- リスタート情報更新 -->
<insert id="INSERT_JOB_RESTART_POINT" parameterClass="jobRestartInfo">
  INSERT INTO
    RESTART_CONTROL (REQUEST_NO , JOB_ID , PARTITION_NO , PARTITION_KEY ,
                     RESTART_POINT , JOB_CONTEXT , STATE , UPDATE_TIME ,
                     REGISTER_TIME)
    VALUES(#requestNo# , #jobId# , #partitionNo# , #partitionKey# ,
           #restartPoint# , #jobContext:BLOB# , #state#   , current_timestamp ,
           current_timestamp)
</insert>
```

ジョブコンテキストは BLOB を明示的にする

ジョブコンテキストは BLOB を明示的にする

■ 関連機能

- 『BA-01 トランザクション管理機能』
- 『BD-01 ビジネスロジック実行機能』
- 『BD-02 対象データ取得機能』
- 『BE-01 同期型ジョブ起動機能』

■ 使用例

- 機能網羅サンプル(functionsample-batch)
- チュートリアル(tutorial-batch)

■ 備考

- なし。

BE-05 処理結果ハンドリング機能

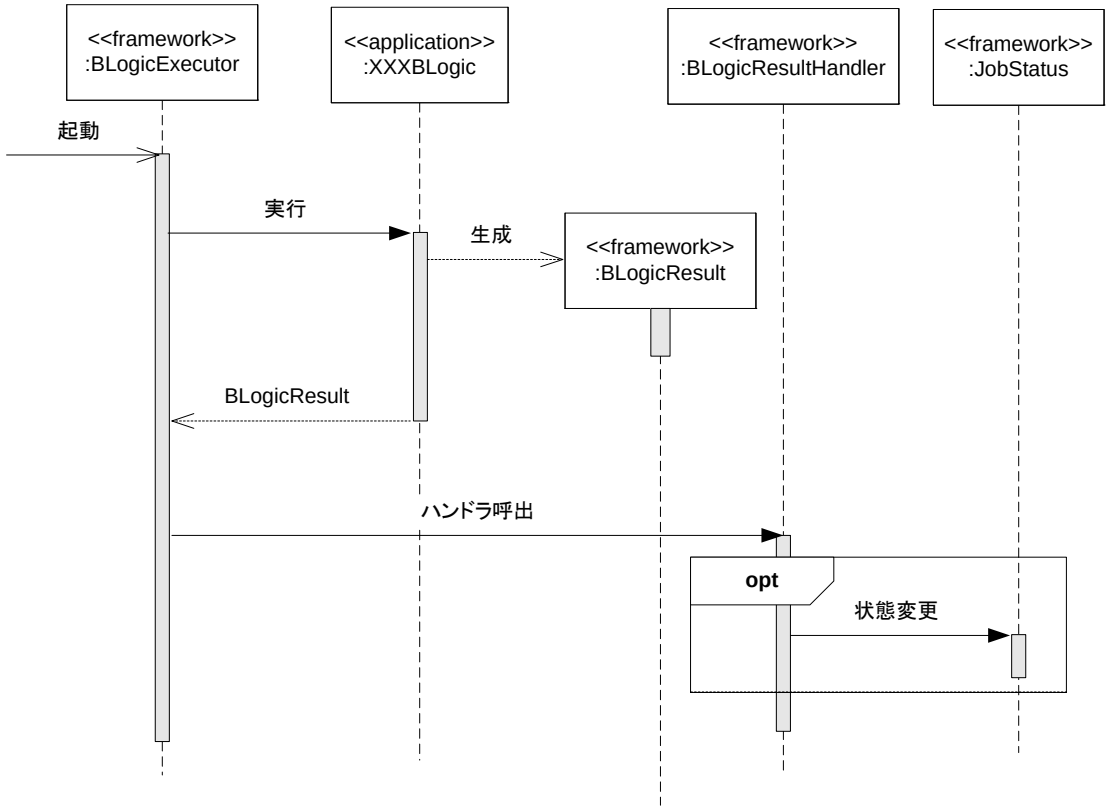
■ 概要

◆ 機能概要

- ジョブを構成する処理（ビジネスロジックやジョブ前処理など）の処理結果を、ジョブ全体の処理結果に反映する機能である。
  - たとえば、個々の処理の処理結果がエラーであった場合に、そのエラー結果をジョブ全体の処理結果に反映することでジョブを終了する。
- 処理結果ハンドラインタフェース並びに実装クラスを提供する。

◆ 概念図

- 処理結果ハンドラの位置付け
  対象とする処理（ビジネスロジック等）の処理結果は、その処理結果に対応する処理結果ハンドラ（ビジネスロジック処理結果ハンドラ等）によって処理される。



◆ 解説

- 提供インタフェースと実装クラス
  フレームワークでは以下のインタフェースと実装クラスを提供する。

項番	処理結果	インタフェース 実装クラス
1	ビジネスロジック処理結果	jp.terasoluna.fw.batch.core.BLogicResultHandler
		jp.terasoluna.fw.batch.standard.StandardBLogicResultHandler jp.terasoluna.fw.batch.springsupport.transaction.TransactionalBLogicResultHandler
2	対象データ取得処理結果	jp.terasoluna.fw.batch.core.CollectorResultHandler
		jp.terasoluna.fw.batch.standard.StandardCollectorResultHandler
3	JDBC バッチ更新処理結果	jp.terasoluna.fw.batch.core.BatchUpdateResultHandler
		jp.terasoluna.fw.batch.standard.StandardBatchUpdateResultHandler
4	サポートロジック(ジョブ前処理、先頭チャンク前処理、ジョブ後処理、最終チャンク後処理)処理結果	jp.terasoluna.fw.batch.core.SupportLogicResultHandler
		jp.terasoluna.fw.batch.standard.StandardSupportLogicResultHandler jp.terasoluna.fw.batch.springsupport.transaction.TransactionalSupportLogicResultHandler

- ビジネスロジック処理結果ハンドラの処理
  - BLogicResult のリターンコードが"NORMAL_END"または“ERROR_END”である場合、ジョブステータスのジョブ状態を終了状態に変更し、BLogicResult に設定されたジョブ終了コードをジョブステータスに反映する。リターンコードについては、『BD-01 ビジネスロジック実行機能』参照のこと。
  - BLogicResult には、フレームワークへ JDBC バッチ更新を依頼する場合に、更新対象の SQL ID およびパラメータオブジェクトが登録される。StandardBLogicResultHandler では、BLogicResult に登録されたバッチ更新情報を取得し、JDBC バッチ更新リストに追加する。詳細は『BB-01 データベースアクセス機能』参照のこと。
  - BLogicResult のリターンコードが“ERROR_END”である場合、エラーデータのログ出力を行う。
  - TransactionalBLogicResultHandler は、ビジネスロジック処理結果に対応したトランザクション処理を行う実装クラスである。

- 対象データ取得処理結果ハンドラの処理
  Collector の処理結果である CollectorResult を処理する。CollectorResult のリターンコードが“ERROR_END”である場合、エラーデータのログ出力を行う。

- JDBC バッチ更新処理結果ハンドラの処理
  バッチ更新成功時にのみ呼ばれる。バッチ更新が成功したバッチ更新情報をジョブステータスに反映する。

- サポートロジック処理結果ハンドラの処理
  サポートロジックが返却する BLogicResult のリターンコードが”NORMAL_END”または“ERROR_END”である場合、ジョブステータスのジョブ状態を終了状態に変更し、BLogicResult に設定されたジョブ終了コードをジョブステータスに反映する。
  また、“ERROR_END”である場合にはログ出力を行う。

- ジョブ終了コードについて
  - ジョブ終了コードはビジネスロジックから任意で指定することができる。ビジネスロジックでジョブ終了コードを指定する方法の詳細は、『BD-01 ビジネスロジック実行機能』を参照のこと。

  - Java VM が異常終了した場合に返却する終了コードは’1’となる。Java VM が異常終了する主な原因は、ジョブ起動時に行われるパラメータデータ初期化処理異常、DI コンテナ生成失敗、クリティカルなエラーをフレームワークで処理しきれなかった場合などが挙げられる。ビジネスロジックでジョブ終了コードに’1’を指定した場合、ビジネスロジックで指定した終了コードと Java VM が異常終了した場合に返却する終了コードとの区別がつかなくなるため、避けたほうが良い。

  - ビジネスロジックでジョブ終了コードに’1’を設定している場合は、-D オプションを利用し任意の終了コードに置換することで、終了コードによって JavaVM の異常終了を検知することができる。-D オプションによるジョブ終了コード置換処理は正常に処理が行われた際にのみ実行される。フレームワークが例外を処理しきれない場合は実行されない。ジョブ終了コードを置換する方法の詳細は、『BE01-同期型ジョブ起動機能』を参照のこと。

  - ジョブ終了コード’1’については、必要があればジョブを起動するシェルスクリプト等によってハンドリングを行うこと。上述の通り Java VM が異常終了すると終了コード’1’が返却されるため、ジョブスケジューラなどの設定次第ではジョブが正常終了と判断される場合がある。その際にはジョブを起動するシェルスクリプト等でジョブ終了コードを取得し、適宜ハンドリングする必要が生じる。
    シェルスクリプトによる終了コードの取得については『BE-01 同期型ジョブ起動機能』を参照のこと。

  - ビジネスロジックでジョブ終了コードを指定しなかった場合（BlogicResult にジョブ終了コードが設定されていない場合）には、ジョブのステータスに応じて以下のデフォルトのジョブ終了コードを返却される。

項番	ジョブのステータス	ジョブ終了コード
1	正常終了(ENDING_NORMALLY)	0
2	異常終了(ENDING_ABNORMALLY)	100
3	中断/強制終了(SUSPENDING)	200

異常終了（ENDING_ABNORMALLY）は、ビジネスロジックやサポートロジックが ERROR_END を返却した場合、あるいは例外が発生した場合のジョブのステータスである。ERROR_END が返却された場合にジョブのステータスを異常終了に変更する処理は、StandardBLogicResultHandler で行われる。例外が発生した場合にジョブのステータスを異常終了に更新する処理は、例外ハンドラで行われる。例外ハンドラについての詳細は、『BH-01 例外ハンドリング機能』を参照のこと。
Collector で取得した対象データが 0 件であった場合には、StandardCollectorResultHandler では正常終了として扱う。
上記のデフォルト終了コードは「DefaultValueBean.xml」上の定義を編集することで、プロジェクトの要件に合わせて変更することができる。

- 同期型ジョブ起動の場合はジョブスケジューラにジョブ終了コードを返却し、非同期型ジョブ起動の場合はジョブ終了コードをジョブ管理テーブルに登録する。同期型ジョブ起動および非同期型ジョブ起動については『BE-01 同期型ジョブ起動機能』、『BE-02 非同期型ジョブ起動機能』を参照のこと。

■ 使用方法

◆ コーディングポイント

- フレームワーク Bean 定義ファイルの設定

必要なインタフェースを実装したクラスを指定する。

```
<!-- BlogicResult のハンドラ -->
<bean id="BLogicResultHandler"
 class="jp.terasoluna.fw.batch.standard.StandardBLogicResultHandler"/>
<!-- CollectorResultのハンドラ -->
<bean id="CollectorResultHandler"
 class="jp.terasoluna.fw.batch.standard.StandardCollectorResultHandler"/>
<!-- BatchUpdateResultのハンドラ -->
<bean id="batchUpdateResultHandler"
 class="jp.terasoluna.fw.batch.standard.StandardBatchUpdateResultHandler"/>
<!-- SupportLogicResult のハンドラ -->
<bean id="supportLogicResultHandler"
 class="jp.terasoluna.fw.batch.standard.StandardSupportLogicResultHandler"/>
```

- ジョブ終了コードの設定（DefaultValueBean.xml）

ジョブのステータスをマップのキーとして、そのジョブステータスでジョブが終了した場合のデフォルトのジョブ終了コード（数値）を設定する。
キーとなるステータスには、以下の３つを指定する。
・ENDING_NORMALLY
・ENDING_ABNORMALLY
・SUSPENDING

```
<!-- 終了コードの定義 -->
<util:map id="exitCodeMap">
    <entry key="ENDING_NORMALLY" value="0"/>
    <entry key="ENDING_ABNORMALLY" value="100"/>
    <entry key="SUSPENDING" value="200"/>
</util:map>
```

◆ 拡張ポイント

- ジョブ終了コードの拡張
  フレームワークが提供する処理結果ハンドラの実装クラスでは、デフォルトのジョブ終了コードを返却する。ジョブ終了コードを追加・変更する場合や、ジョブ終了コードに対する処理を追加・変更する場合は、StandardBLogicResultHandler および StandardSupportLogicResultHandler の拡張を行う。

- ログ出力
  処理結果ハンドラの各実装クラスを拡張、あるいは置き換えを行って、プロジェクトの要件に合わせたログを出力するように拡張することがきる。

- ジョブステータスの変更

ジョブステータスを変更する場合は、基本的には ENDING_NORMALLY, ENDING_ABNORMALLY, 変更しないの３つから選択を行い、その他のステータスへの変更は行わないようにする。特に起動前の状況である SUBMITTED に変更することは禁止する。

■ 関連機能

- 『BB-01 データベースアクセス機能』
- 『BD-01 ビジネスロジック実行機能』
- 『BE-01 同期型ジョブ起動機能』
- 『BE-02 非同期型ジョブ起動機能』

■ 使用例

- 機能網羅サンプル(functionsample-batch)
- チュートリアル(tutorial-batch)

■ 備考

- なし。

BF-01 メッセージ管理機能

■ 概要

◆ 機能概要

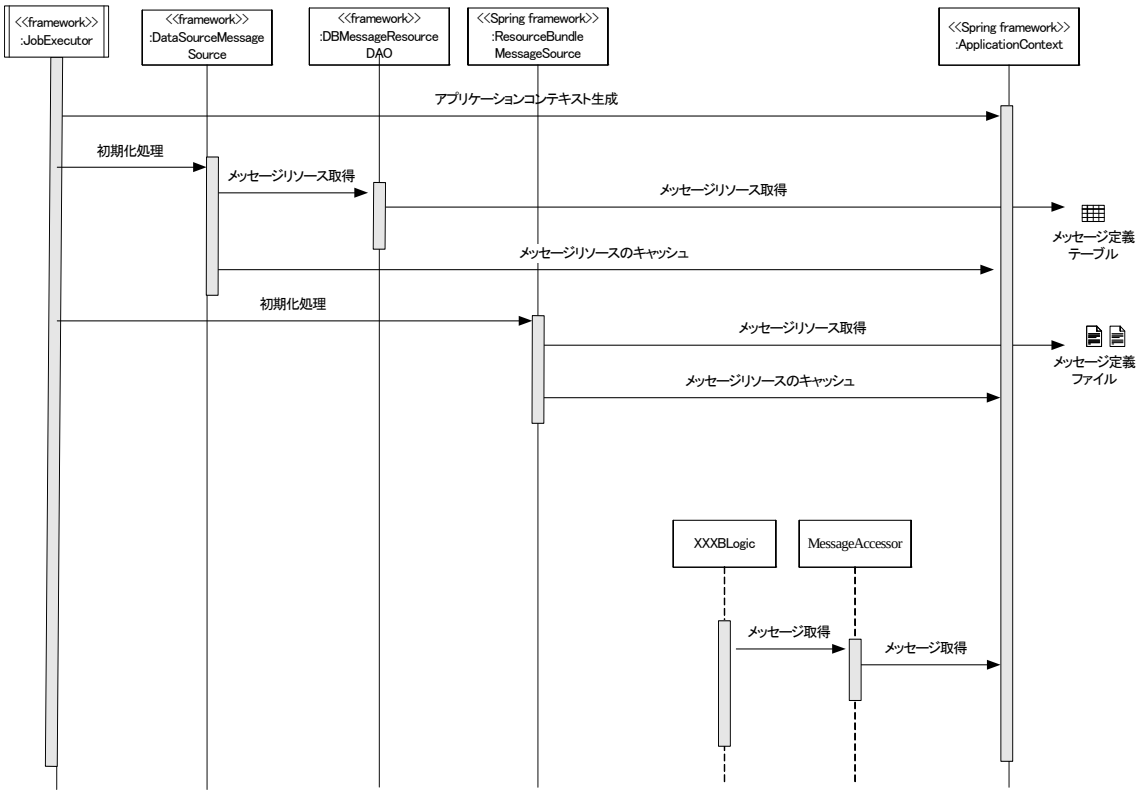
- ログ出力等で利用する文字列を管理するための、メッセージ管理機能を提供する。メッセージリソースとして、ファイルメッセージリソースと、ＤＢメッセージリソースが利用できる。
  - ファイルメッセージリソースは、Spring が標準で提供する MessageSource 機能を利用している。TERASOLUNA-Batch ではデフォルトの設定として、業務共通メッセージリソース定義ファイルをファイルメッセージリソースとして定義している。本ファイルには業務アプリケーションで利用するメッセージを定義する。
  - ＤＢメッセージリソースは、データベース内のメッセージ定義テーブルでメッセージを定義することが可能である。ＤＢメッセージリソースを利用するためのクラスを TERASOLUNA-Batch で提供する。
  - ファイルメッセージリソースとＤＢメッセージリソースを共用することが可能である。

- システムエラーメッセージ、ファイル入力チェックエラーメッセージ等、TERASOLUNA- Batch が規定するデフォルトメッセージを提供する。
  - システムメッセージリソース定義ファイルを提供し、デフォルトのメッセージを定義している。本ファイルは TERASOLUNA-Batch の『BE-05 処理結果ハンドリング機能』『BH-01 例外ハンドリング機能』において、フレームワークの実装クラスを拡張し、プロジェクトの要件に合わせたログを出力する際に使用することができる。

- ビジネスロジック等からメッセージを取得するために利用する、メッセージ取得用のデフォルト実装を提供する。

◆ 概念図

- メッセージリソース初期化処理及びメッセージ取得の流れ
  メッセージリソース初期化処理では、フレームワーク Bean 定義の設定に従い、ファイル及びデータベースに定義されたメッセージリソースを取り込み、アプリケーションコンテキスト(フレームワーク BeanFactory)に保持する。



ビジネスロジックからのメッセージ取得は、メッセージ取得用クラスを介して、アプリケーションコンテキストに保持されたメッセージにアクセスする。

◆ 解説

- ファイルメッセージリソース
  - 業務共通メッセージリソース定義ファイル（application-messages.properties）
    ビジネスロジック等から取得する業務メッセージを定義する。

```
error.UC02.00001=カレンダーに存在しない年月日です。
error.UC02.00002=期間が不正です。開始年月日={0}  終了年月日={1}
    ・
    ・
```

    複数のファイルに分割してメッセージを定義することも可能である。

  - システムメッセージリソース定義ファイル（system-message.properties）
    システムエラーメッセージ、ファイル入力チェックエラーメッセージ等、フレームワークがログ出力のために取得するメッセージが定義されている。

```
errors.8004C010=DB 接続エラーが発生しました:エラー詳細={0}
errors.8004C011=CSV ファイル読込みエラーが発生しました : エラー詳細={0}
    ・
errors.alphaNumericString=半角英数字チェックエラーが発生しました:項目名={0}.
    ・
```

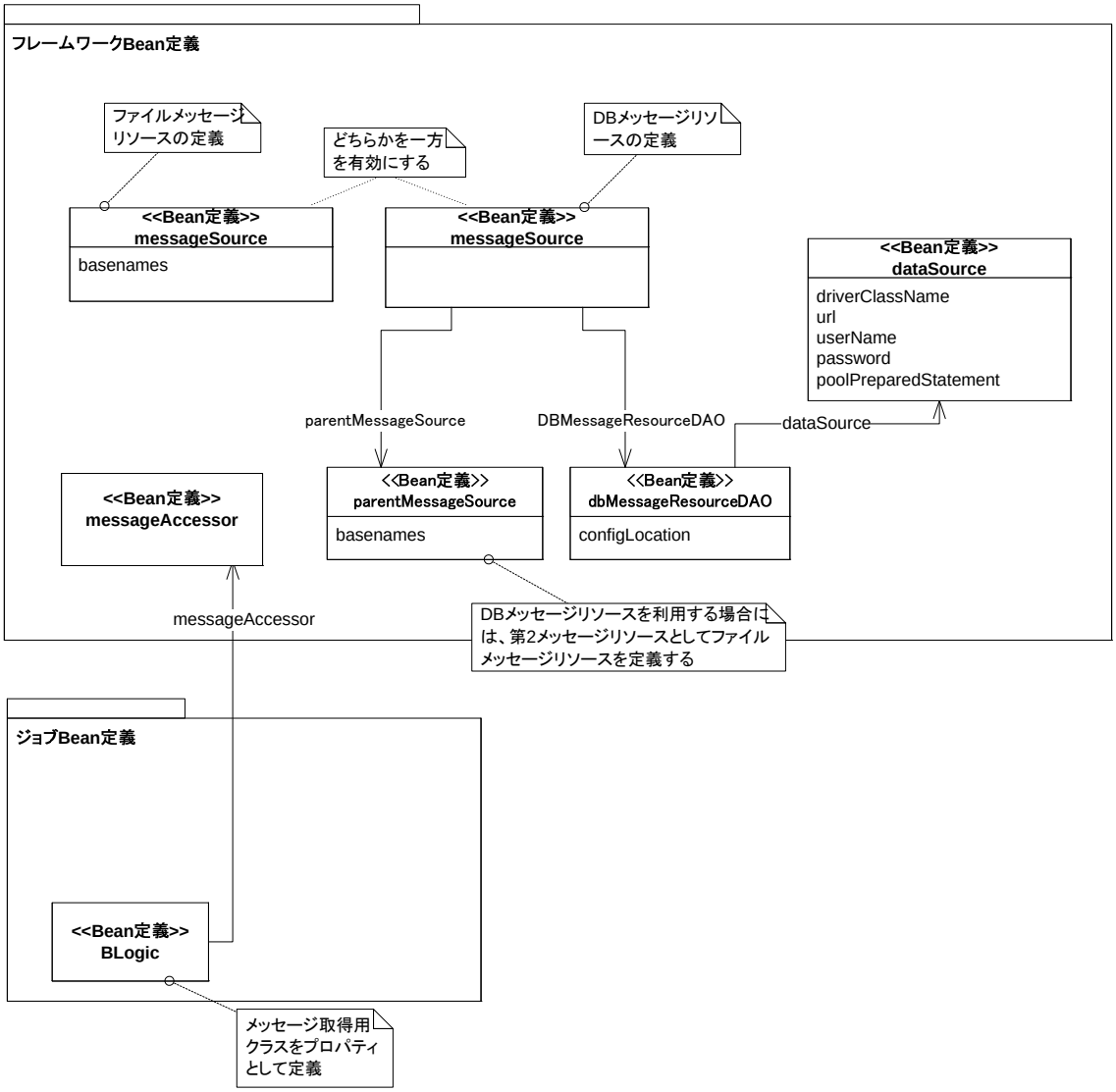
- DB メッセージリソース
  - メッセージ定義テーブル
    メッセージ文字列はデータベース内に以下のテーブルを作成して格納する。
    ◆テーブル名：MESSAGES

カラム名	説明	格納データ例
CODE	メッセージコードを格納する	error.UC01.00001
MESSAGE	メッセージ本文を格納する	カレンダーに存在しない年月日です。

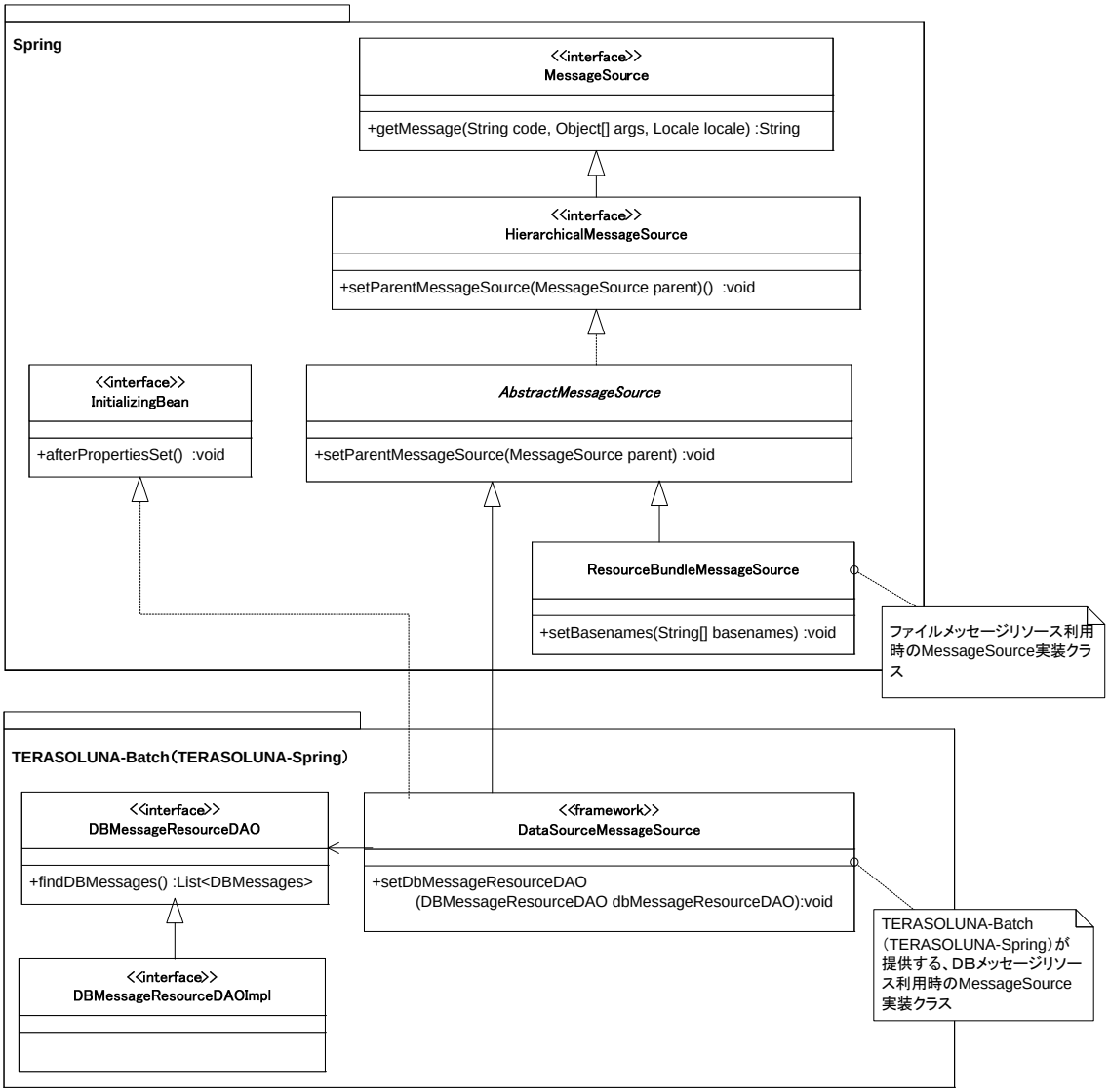
    ・ CODE を PrimaryKey とする。

    なお、上記は一例であり、テーブル名及びカラム名は各プロジェクトで自由に変更できる。

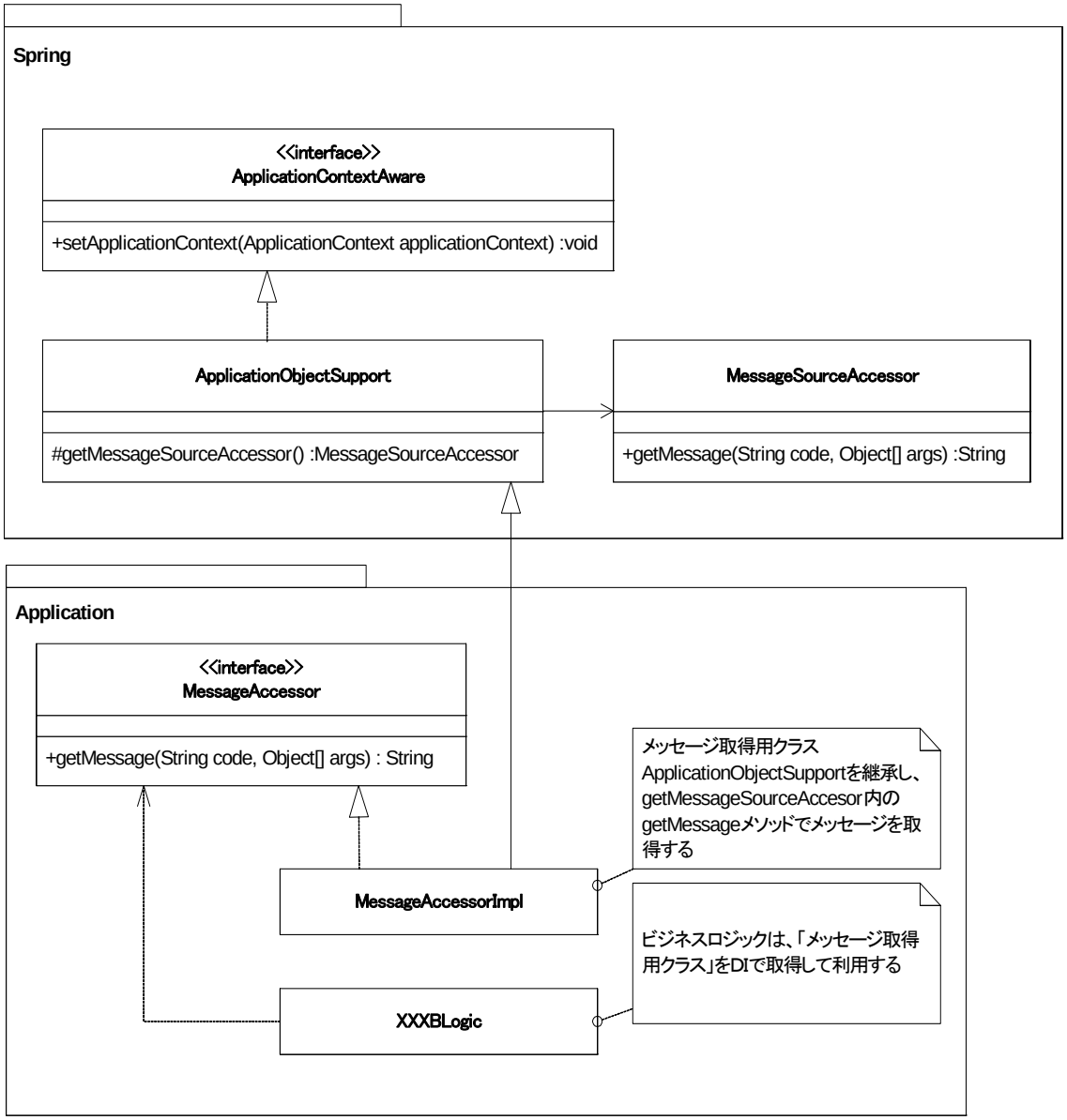
- Spring Bean 定義ファイルの構造
  フレームワーク Bean 定義ファイルには、ファイルメッセージリソース用の Bean 定義とＤＢメッセージリソース用の Bean 定義が雛形として設定されている。利用するメッセージリソースに合わせた設定を行なうこと。
  また、サンプル実装のメッセージ取得用クラスの Bean 定義が設定されている。ジョブの定義を行う際、ビジネスロジックのプロパティにセットすること。



- メッセージ管理のクラス構造（メッセージリソースの管理）
  ファイルメッセージリソース利用時は ResourceBundleMessageSource クラスが、DBメッセージリソース利用時は DataSourceMessageSource クラスが、メッセージリソースのメッセージを取り出し、アプリケーションコンテキストに保持する。



- メッセージ管理のクラス構造（メッセージの取得処理）
  アプリケーションコンテキストに保持されているメッセージを取得する際には、org.springframework.context.support.ApplicationObjectSupport に定義されている MessageSourceAccesor 内の getMessage メソッドを使用する。（詳細については spring フレームワークの API を参照のこと。）
  なお、TERASOLUNA-Batch では、メッセージ取得用クラスとして MesageAccesor および MesageAccesorImpl を提供している。



- ＤＢメッセージソースを利用するための提供クラス
  - ＤＢメッセージリソースを利用するためのクラスを提供する。（TERASOLUNA-Spring で提供されているクラスを利用）

項番	クラス名	概要
1	jp.terasoluna.fw.message.DataSourceMessegeSource	メッセージ定義テーブルを参照し、メッセージの生成、発行を行なうクラス

  - DB メッセージリソース取得用の DAO インタフェース及び、デフォルト実装を提供する。（TERASOLUNA-Spring で提供されているクラスを利用）

項番	インタフェース名／クラス名	概要
1	jp.terasoluna.fw.message.DBMessageResourceDAO	メッセージ定義テーブルにアクセスするための DAO インタフェース
2	jp.terasoluna.fw.message.DBMessageResourceDAOImpl	メッセージ定義テーブルにアクセスするための DAO の実装クラス

  - DBMessageResourceDAOImpl が発行する SQL は以下のとおり。

SELECT CODE,MESSAGE FROM MESSAGES

- ビジネスロジックからメッセージの取得を行なうための提供クラス
  - メッセージ取得用クラスとして、下記のインタフェースと実装クラスを提供する。

項番	インタフェース名／クラス名	概要
1	jp.terasoluna.fw.batch.messages.MessageAccessor	メッセージ取得用インタフェース
2	jp.terasoluna.fw.batch.springsupport.messages.MessageAccessorImpl	メッセージ取得用実装クラス

  - 取得したいメッセージのメッセージ ID およびプレースホルダに埋め込む文字列を引数に指定して呼び出すと、該当メッセージを返却する。
  - メッセージリソース中にプレースホルダ（「{0} は必須項目です）の {0}）を定義しておくことで、引数に指定した文字列を動的に埋め込んだメッセージを取り出すことができる。

- メッセージリソース初期化時およびメッセージ取得時のエラーについて

  メッセージリソースの初期化時にエラーが発生した場合には、同期型ジョブ起動ではジョブを終了する。非同期型ジョブ起動では非同期バッチデーモンを終了する。

  同期型ジョブ起動、非同期型ジョブ起動の詳細については『BE-01 同期型ジョブ起動機能』『BE-02 非同期型ジョブ起動機能』を参照のこと。

  ビジネスロジック等からメッセージを取得する際に、当該メッセージ ID の定義が無い等の理由でメッセージの取得に失敗した場合には、メッセージ取得用クラスのデフォルト実装では、メッセージ ID を返却する。

■ 使用方法

◆ コーディングポイント

- フレームワーク Bean 定義ファイルの設定例
  “messageSource”という識別子の Bean を設定することで、メッセージ管理機能を利用することができる。

➢ ファイルメッセージリソースを利用する場合

```
<bean id="messageSource"
  Class="org.springframework.context.support.ResourceBundleMessageSource">
     <property name="basenames"
        value="application-messages,system-messages"/>
</bean>
```

ResourceBundleMessageSource を指定する。

利用するファイルメッセージリソースを指定する。

➢ ＤＢメッセージリソースを利用する場合

```
<bean id="messageSource"
  class="jp.terasoluna.fw.message.DataSourceMessageSource">
     <property name="parentMessageSource"
        ref bean="parentMessageSource"/>
     <property name="dbMessageResourceDAO"
        ref bean="dbMessageResourceDAO"/>
</bean>

<bean id="parentMessageSource"
  class="org.springframework.context.support.ResourceBundleMessageSource">
     <property name="basenames"
        value="application-messages,system-messages"/>
</bean>

<bean id="dbMessageResourceDAO"
  class="jp.terasoluna.fw.message.dbMessageResourceDAOImpl">
    <property name="dataSource" ref bean="dataSource"/>
    <property name="tableName" value="MESSAGES"/>
    <property name="codeColumn" value="CODE"/>
    <property name="messageColumn" value="MESSAGE"/>
</bean>

<bean id="dataSource"    ·····>
</bean>
```

DataSourceMessageSource を指定する。

ParentMessageSource を指定する。

DB メッセージ取得用の DAO を指定する。

DAO の実装を指定する。

dataSource を指定する。

✧ “parentMessageSource”の指定により、第 2 メッセージリソースを追加している。”messageSource”内にメッセージが存在しなかった場合、第 2

メッセージリソースで指定したメッセージリソース内を検索する。

- ✧ "dbMessageResourceDAO"の bean 定義の property 設定により、メッセージ定義テーブルのテーブル名及びカラム名を指定する。
- ✧ “dataSource”の Bean 定義については、『BB-01 データベースアクセス機能』を参照のこと。

- ビジネスロジックからのメッセージ取得方法

ビジネスロジックからのメッセージ取得については、メッセージ取得用の業務共通クラスを利用することを推奨する。TERASOLUNA-Batch ではメッセージ取得用クラスを提供している。

- メッセージ取得用クラスインタフェース

jp.terasoluna.fw.batch.messages.MessageAccessor

```
public interface MessageAccessor {
    String getMessage(String code, Object[] args);
}
```

- メッセージ取得用クラス実装クラス

jp.terasoluna.fw.batch.springsupport.messages.MessageAccessorImpl

```
public class MessageAccessorImpl extends ApplicationObjectSupport
implements MessageAccessor {

    /**
    * メッセージキーで指定したメッセージを取得する。
    * 指定されたメッセージIDに対応するメッセージが存在しない場合には
    * メッセージIDを返却する。
    */
    public String getMessage(String code, Object[] args) {

       // デフォルトメッセージ(デフォルトメッセージとしてメッセージIDを設定)
       String defaultMessage = code;

      // メッセージを返却する
      return getMessageSourceAccessor().getMessage(code,
                                                   args,
                                                   defaultMessage);
    }
}
```

org.springframework.context.support.ApplicationObjectSupport に定義されている MessageSourceAccesor 内の getMessage メソッドを使用する。

➢ ビジネスロジック（サンプル）

```
public class XXXBLogic implements BLogic<XXXInputData, XXXJobContext> {

    // メッセージ取得クラス用setter
    MessageAccessor messageAccessor = null;
    public void setMessageAccessor(MessageAccessor msgAcc) {
        this.messageAccessor = msgAcc;
    }

   // ビジネスロジック
   Public BLogicResult execute(XXXInputData inputData
                               XXXJobContext jobContext) {
        String outPutMessage = null;
            ・・・省略・・・

        outPutMessage = messageAccessor.getMessage("msg.00101", sectionCode);
            ・・・省略・・・
    }
}
```

メッセージ取得用クラスを DI するためのセッターを記述する。

メッセージ取得用クラスのメッセージ取得メソッドを利用する。

➢ Bean 定義ファイルの設定例

```
<bean id="messageAccessor"
Class="jp.terasoluna.fw.batch.springsupport.messages.MessageAccessorImpl"/>
```

```
<bean id="bLogic"
   Class="jp.terasoluna.fw.batch.service.XXXBlogic"/>
      <property name = "messageAccessor" ref=" messageAccessor" />
</bean>
```

◆ 拡張ポイント

- ＤＢメッセージリソース用のメッセージ定義テーブルのスキーマの変更
  TERASOLUNA-Spring で規定しているメッセージ定義テーブル及び DB メッセージ取得用 DAO はデフォルト実装であり、プロジェクトにあわせて変更することが可能である。
  変更したテーブルスキーマにあわせて、DB メッセージ取得 DAO および SQL の変更、フレームワーク Bean 定義ファイルの変更を行なう。

- メッセージの国際化対応
  ファイルメッセージリソースについては、Spring が標準で提供する MessageSource 機能が国際化に対応している。詳細については Spring フレームワークの API を参照のこと。
  ＤＢメッセージリソースについては、TERASOLUNA-Spring が提供するＤＢメッセージ機能が国際化に対応している。詳細については TERASOLUNA Server Framework for Java（Rich 版）の『RG-01 ＤＢメッセージ機能』を参照のこと。

■ 関連機能

- 『BB-01 データベースアクセス機能』
- 『BE-01 同期型ジョブ起動機能』
- 『BE-02 非同期型ジョブ起動機能』
- 『BE-05 処理結果ハンドリング機能』
- 『BH-01 例外ハンドリング機能』
- TERASOLUNA Server Framework for Java（Rich 版）『RG-01 ＤＢメッセージ機能』

■ 使用例

- 機能網羅サンプル(functionsample-batch)
- チュートリアル(tutorial-batch)

■ 備考

- なし

BH-01 例外ハンドリング機能

■ 概要

◆ 機能概要

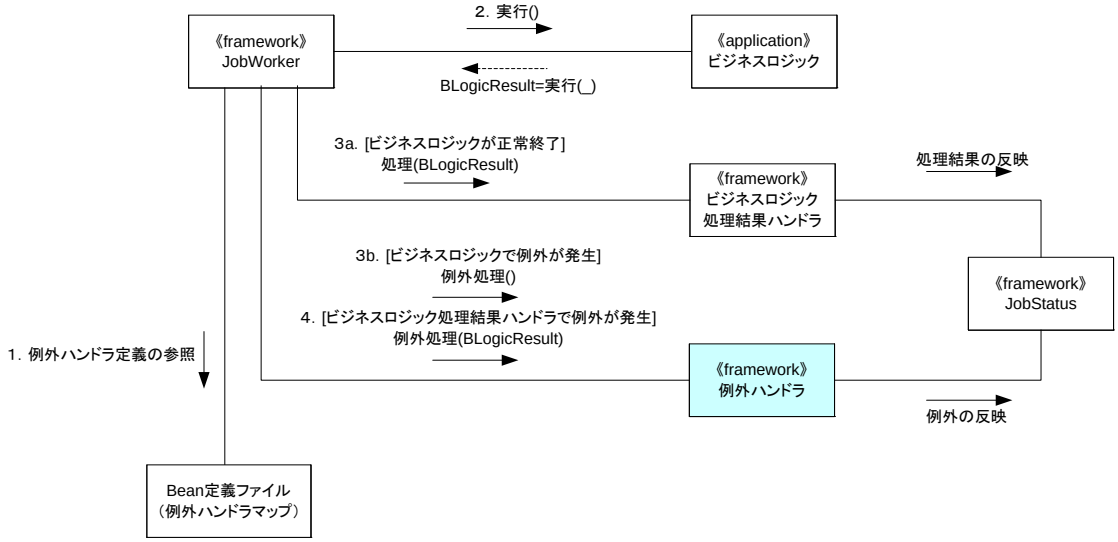
- 例外ハンドリング機能では、アプリケーション（ビジネスロジック、ジョブ前処理など）で発生した例外を例外ハンドラで処理する機能を提供する。
  - フレームワークは例外ハンドラのインタフェースを規定し、デフォルト実装を提供する。フレームワークのデフォルト実装を拡張する、あるいは置き換えることで例外処理を変更することができる。

- アプリケーションで発生した例外は、フレームワークが提供する例外クラスでラップされて処理される。
  - フレームワークが提供する例外クラスは、例外発生元処理の種類毎に異なるクラスが提供されており、例外発生元処理の種類に対応した例外クラスでラップされる。

- フレームワークは例外ハンドラマップに従って例外ハンドラを起動する。
  - 例外ハンドラマップは、フレームワークで提供する例外クラスをキーとして、その例外が発生した際に起動する例外ハンドラを定義する。

◆ 概念図

- 例外処理の流れの概要（ビジネスロジック例外の場合）



◆ 解説

- 例外ハンドラのインタフェース
  例外ハンドラは、フレームワークで規定するインタフェースを実装して作成する。

項番	インタフェース	概要
1	jp.terasoluna.fw.batch.core.JobExceptionHandler	フレームワークで規定する例外ハンドラのインタフェース

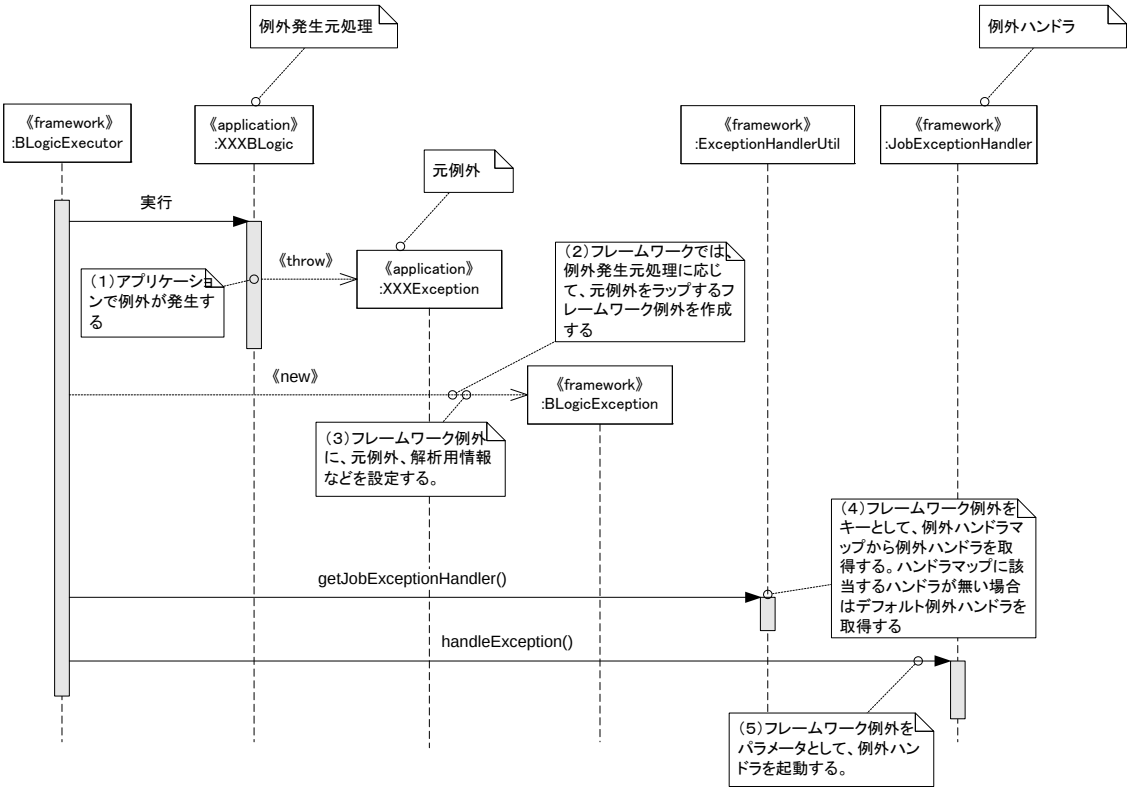
- 例外ハンドラのデフォルト実装
  例外ハンドラは、フレームワークがデフォルト実装を提供する。ログのフォーマット変更等のプロジェクト固有の要件に対応する場合には、ソフトウェアアーキテクトがデフォルト実装の拡張、あるいは置き換えを行うこと。
  フレームワークのデフォルト実装では、ログを出力し、ジョブステータスをエラー終了に更新してジョブを終了する。例外発生元処理のトランザクションはロールバックされる。

- 例外ハンドラマップ
  例外ハンドラマップには、フレームワーク例外クラスをキーとして、例外ハンドラを登録する。
  例外ハンドラマップには、登録順も重要であることに留意すること。例外ハンドラの登録順序は、try-catch を記述する順序で記述しなくてならない。たとえば、ジョブ例外 JobException は、ビジネスロジック例外 BLogicException の親クラスであるため、例外ハンドラマップにおいてビジネスロジック例外よりもジョブ例外を先に登録すると、ビジネスロジック例外が発生した場合であっても常にジョブ例外に対する例外ハンドラが起動される。

- デフォルトの例外ハンドラマップ
  デフォルトの例外ハンドラマップは、JobException に対するハンドラが登録されている。例外が発生した場合には、デフォルト例外ハンドラ定義で定義されている例外ハンドラが起動される。

- デフォルト例外ハンドラ定義
  発生した例外に対する例外ハンドラが例外ハンドラマップに登録されていない場合には、デフォルト例外ハンドラ定義に従って例外ハンドラが起動される。
  デフォルト例外ハンドラ定義は、例外ハンドラマップとは別に定義する。デフォルト例外ハンドラ定義は、個別のジョブで例外ハンドラマップを上書き定義する場合に、例外ハンドラの定義漏れを防ぐための定義である。

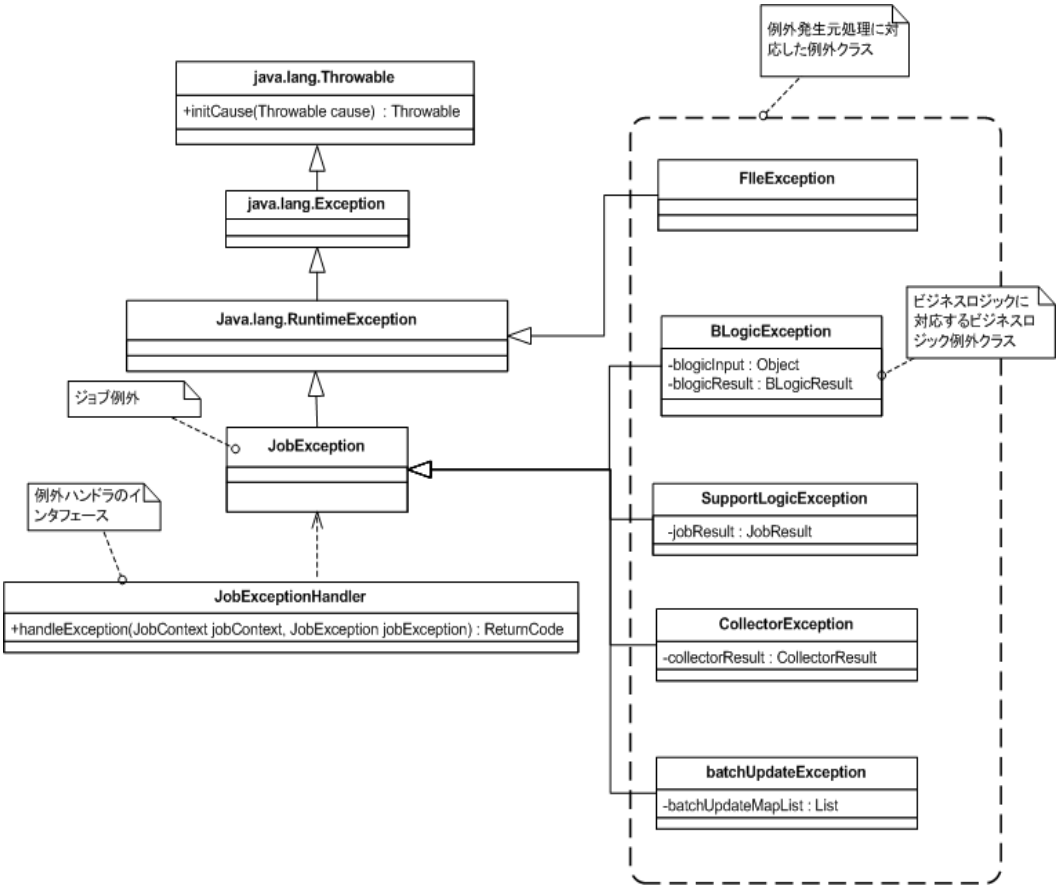
- 例外処理の流れ
  例外が発生した場合には、発生した例外は例外発生元処理に対応したフレームワーク例外クラスによってラップされ、解析に必要な情報がセットされたあと、例外ハンドラマップに従って例外ハンドラが起動される。



（1） アプリケーションで例外が発生し、フレームワークまで伝播する。
（2） フレームワークでは、どの処理から例外がスローされたのかを判定し、対応するフレームワーク例外を作成する。
（3） フレームワークは、作成したフレームワーク例外に、元例外、および解析情報を設定する。
（4） フレームワークは、フレームワーク例外に対応した例外ハンドラを取得する。
（5） フレームワークは、作成したフレームワーク例外をパラメータとして、例外ハンドラを起動する。

※ ただし、初期化例外は起動中に発生するため、例外ハンドラマップに登録した場合・登録しなかった場合のどちらの場合についても例外ハンドラは起動できないため、例外ハンドラを利用することができない。

- 例外ハンドラの入力
  例外ハンドラには、入力として例外発生元処理に対応した例外情報が渡される。例外情報は、例外発生元の処理の種類に対応したクラス階層になっている。例外情報のクラス階層の親クラスは、「ジョブ例外」であり、例外ハンドラのインタフェースでは「ジョブ例外」が入力パラメータとして宣言されている。



  例外クラス階層中のそれぞれの例外クラスは、対応する例外発生元処理に応じて解析用情報を保持する。たとえば、ビジネスロジックに対応する例外クラスである「ビジネスロジック例外」クラスでは、ビジネスロジックの入力パラメータを保持する。

項番	例外発生元処理の種類	例外クラス （論理名／物理名）	例外クラスが保持する情報
1	・ビジネスロジック ・ビジネスロジック処理結果ハンドラ	ビジネスロジック例外／ jp.terasoluna.fw.batch.core.BLogicException	・原因例外 ・ビジネスロジック入力オブジェクト ・BLogicResult
2	・ジョブ前処理 ・ジョブ前処理処理結果ハンドラ ・先頭チャンク前処理 ・先頭チャンク前処理処理結果ハンドラ ・ジョブ後処理 ・ジョブ後処理処理結果ハンドラ ・最終チャンク後処理 ・最終チャンク後処理処理結果ハンドラ	サポート処理例外／ jp.terasoluna.fw.batch.core.SupportLogicException	・原因例外
3	・Collector ・Collector 処理結果ハンドラ	Collector 例外／ jp.terasoluna.fw.batch.core.CollectorException	・原因例外 ・CollectorResult
4	・バッチ更新 ・バッチ更新処理結果ハンドラ	バッチ更新例外／ jp.terasoluna.fw.batch.core.BatchUpdateException	・原因例外 ・バッチ更新情報リスト

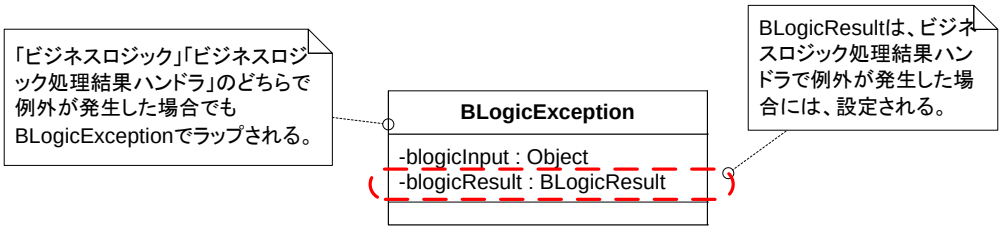
- 例外ハンドラでの処理例
  例外ハンドラでは、入力として渡された例外情報を使って、発生した例外、および例外発生元処理の入力のダンプなどをログに出力することができる。
  また、例外が発生した場合でもその例外を異常系として処理せずに、正常系として処理を進めることができる。たとえば、Collector で FileNotFoundException が発生した（入力ファイルが存在しなかった）際に正常終了したい場合には、フレームワーク例外クラス CollectorException に対する例外ハンドラとして、発生した例外が FileNotFoundException であるときには無視するような例外ハンドラを設定する。

- 例外ハンドラによるジョブステータスへの反映
  例外ハンドラでは、発生した例外に応じてジョブステータスを変更することで、ジョブの継続/終了などを決定することができる。
  たとえば、ビジネスロジックが例外をスローした場合であっても、その例外によって起動された例外ハンドラがジョブステータスを変更しなければ、NORMAL_CONTINUE を返却した場合にはもともとのビジネスロジックが NORMAL_CONTINUE を返却した場合と同じように処理が継続される。
  同様に、例外発生元処理が処理結果ハンドラ（ビジネスロジック処理結果ハンドラ等）である場合にも、例外ハンドラがジョブステータスを更新することで、処理結果ハンドラの処理結果を置き換えることができる。

- 例外ハンドラのトランザクション
  例外ハンドラでトランザクション処理を行うことは推奨されない。
  例外ハンドラでトランザクション処理を行う場合には、例外発生元処理のトランクザクションで処理される。例外ハンドラで例外発生元処理とは別のトランザクション処理を行う場合には、別トランザクションで定義した共通処理を呼び出して処理を行うこと。
  トランザクションモデルが非トランザクションモデルである場合には、例外ハンドラでトランザクション処理を行うことができない。
  トランザクションモデルについては、『BA-01 トランザクション管理機能』を参照のこと。

- 例外ハンドラと処理結果ハンドラとの関係
  例外発生元処理には、ビジネスロジックなどの主たる処理と、主たる処理に対応する処理結果ハンドラの２種類がある。
  これらの２種類の例外発生元処理は、同一のフレームワーク例外クラスで処理される。たとえば、ビジネスロジックで例外が発生した場合であっても、ビジネスロジック処理結果ハンドラで例外が発生した場合であっても、どちらの場合でも元の例外はビジネスロジック例外にラップされて、例外ハンドラが起動される。
  ただし、ビジネスロジックで例外が発生した場合には、BLogicResult が作成されていないためビジネスロジック例外に BLogicResult が設定されていない。それに対してビジネスロジック処理結果ハンドラで例外が発生した場合には、先行するビジネスロジックで BLogicResult が作成されており、ビジネスロジック処理結果ハンドラの入力となっている情報であるため、BLogicResult がビジネスロジック例外に設定される。処理結果ハンドラについては、『BE-05 処理結果ハンドリング機能』を参照のこと。

項番	例外発生元処理の種類	フレームワーク例外クラス	例外クラスが主たる処理の処理結果を持つか
1	ビジネスロジックなどの主たる処理	主たる処理と、その処理の処理結果ハンドラで同一の例外クラス	主たる処理で例外が発生している場合には、主たる処理の処理結果が存在しないため、null が設定される。
2	主たる処理の処理結果ハンドラ		例外クラスに主たる処理の処理結果が設定されている。



- 例外ハンドラで発生した例外の処理
  例外ハンドラから例外がスローされた場合には、フレームワークによって ERROR_ENDが返却された場合と同じように処理される。

■ 使用方法

◆ コーディングポイント

- 例外ハンドラの設定
  フレームワーク Bean 定義ファイルにおいて、ベースジョブ Bean にデフォルト例外ハンドラ定義、および空の例外ハンドラマップが設定されている。
  ジョブ Bean 定義ファイルでは、必要に応じてジョブ固有の例外ハンドラマップを設定する。
  - フレームワーク Bean 定義ファイルの設定例

```
<bean id="defaultJobExceptionHandler"
  class="jp.terasoluna.fw.batch.standard.StandardJobExceptionHandler"/>
   ...
<util:map id="exceptionHandlerMap"">
   <entry>
      <key><bean class="jp.terasoluna.fw.batch.core.JobException"/></key>
      <bean class="jp.terasoluna.fw.batch.standard.StandardJobExceptionHandler"/>
    </entry>
</util:map>
```

デフォルトの例外ハンドラ定義。

デフォルトの例外ハンドラマップ

  - ジョブ Bean 定義ファイルの設定例
    （フレームワーク Bean 定義ファイルでの例外ハンドラ設定を上書きする場合）

```
   ...
  <util:map id="exceptionHandlerMap">
     <entry>
        <key><bean class="jp.terasoluna.fw.batch.core.BLogicException"></key>
         <bean class="jp.terasoluna.....XXXBLogicExceptionHandler"/>
     </entry>
     <entry>
        <key><bean class="jp.terasoluna.fw.batch.core.CollectorException"></key>
        <bean class="jp.terasoluna.....XXXCollectorExceptionHandler"/>
     </entry>
  </util:map>
```

例外ハンドラを登録したいフレームワーク例外をキーとして、例外ハンドラを登録する。
例外ハンドラはフレームワークで規定するインタフェースを実装して作成する。

➢ 例外ハンドラの実装例

```
class XXXBLogicExceptionHandler implements JobExceptionHandler {
    ...

    public void handleException(JobContext jobContext,
                                JobException jobException,
     JobStatus jobStatus) {
        ...
        Throwable t = jobException.getCause();
        if (t instanceof FileNotFoundException) {
            log.debug(jobContext.getJobID(), ...);
            super.handleException(jobContext, jobException, jobStatus);
        }
    }
}
```

例外ハンドラは、フレームワークで規定するインタフェースを実装して作成する。この例では、JobExceptionHandler を実装しているフレームワークのデフォルト実装クラスを拡張している。

ジョブコンテキストやジョブ例外から、解析用の情報を取得し、ログ出力などを行うことができる。

ジョブ個別の例外ハンドリングでない部分は、フレームワークで用意するデフォルト実装の例外ハンドラ、あるいはプロジェクトで用意する例外ハンドラに任せるような実装が望ましい。

◆ 拡張ポイント

- デフォルトの例外ハンドラ実装の拡張
  フレームワークで提供している例外ハンドラ実装はデフォルト実装であり、プロジェクトにあわせてログ出力などを拡張することができる。

- ジョブステータスの変更
  ジョブステータスを変更する場合は、基本的には ENDING_NORMALLY, ENDING_ABNORMALLY, 変更しないの３つから選択を行い、その他のステータスへの変更は行わないようにする。特に起動前の状況である SUBMITTED に変更することは禁止する。

■ 関連機能

- 『BA-01 トランザクション管理機能』
- 『BE-05 処理結果ハンドリング機能』

■ 使用例

- 機能網羅サンプル(functionsample-batch)
- チュートリアル(tutorial-batch)

■ 備考

- なし。

BI-01 Commonj対応機能

■ 概要

◆ Commonj概要

- ワークマネージャを使用したスレッド管理機能を提供する。
  - スレッド管理機能により、ジョブの並列実行機能を提供する。
    詳細は「JSR:237 Work Manager for Application Server」を参照のこと。

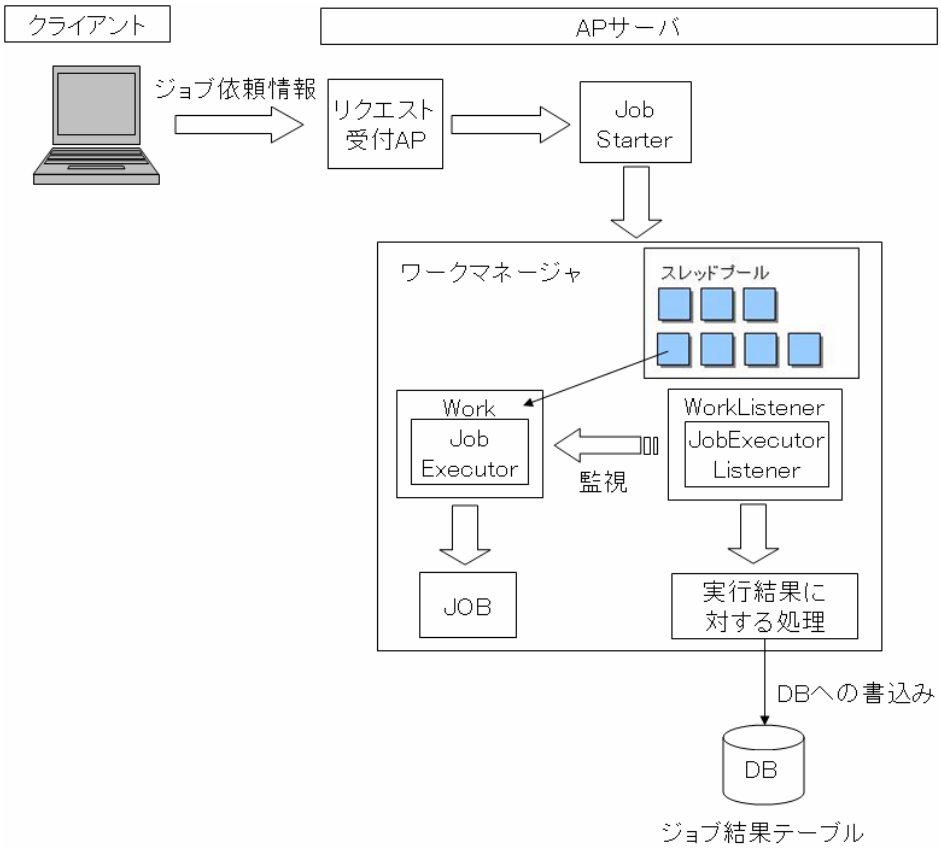
- タイマーマネージャを使用したスケジューリング機能を提供する。
  - java.util.Timer クラスの仕様に代わるアプローチを提供する。
    詳細は「JSR:236 Timer for Application Server」を参照のこと。

◆ 機能概要

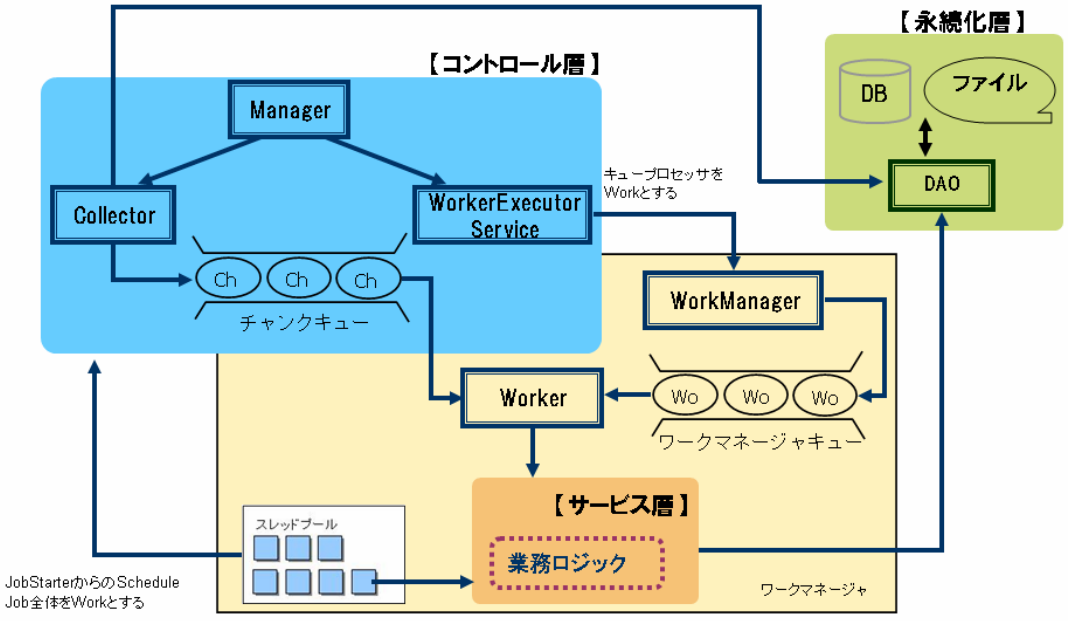
- クライアントから渡されてきたジョブ依頼情報を元に、AP サーバ上で対象ジョブが実行される。
- ワークマネージャがデータをチャンクキューへ詰め込み、スレッドプールからスレッドを割り当て、対象ジョブを実行する。
- 非同期処理を用いて、ジョブ依頼情報を DB に登録する事でジョブが実行される。
  ※ JobStarter を同期型として実装することも可能

◆ 概念図

- ジョブ実行



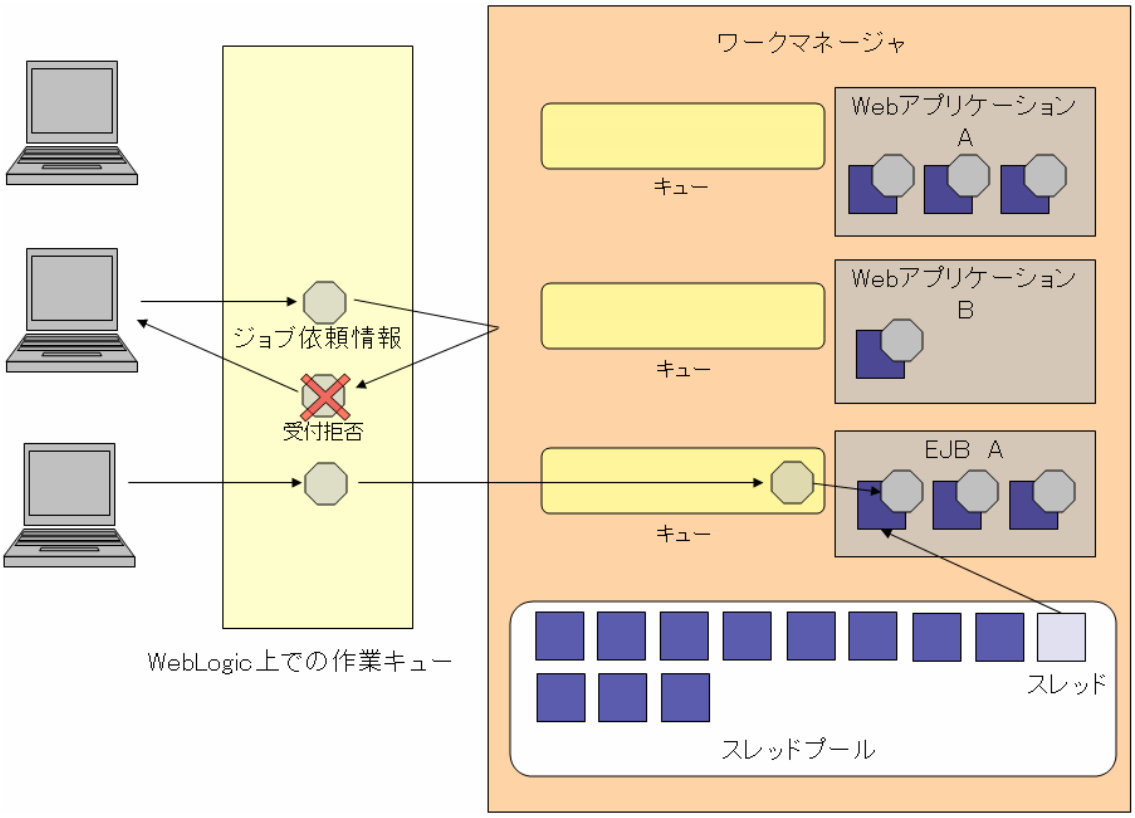
- AP サーバ上でのアーキテクチャ概観



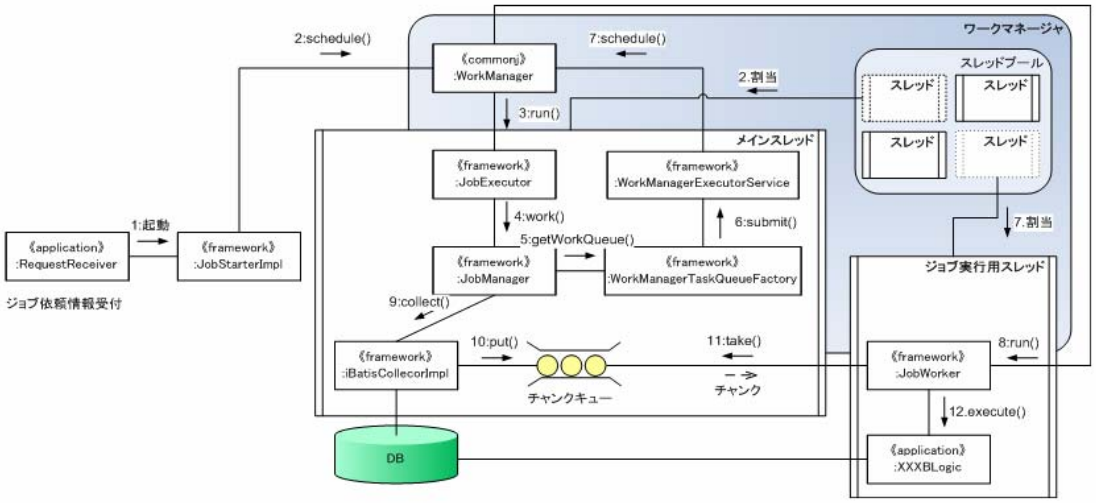
◆ 解説

- AP サーバ上でのジョブ実行の概要
  - ➢ AP サーバを介して行われる処理フローは以下の通りである。
  - (1) ジョブ依頼情報を渡して JobStarter を実行する。
  - (2) JobStarter はワークマネージャに対し、Work を実装した JobExecutor の処理依頼を行う。
  - (3) ワークマネージャは Work にスレッドを割り当て、Work の処理を実行する。
    それによりクライアントから依頼されたジョブが実行される。
    JobStarter は処理依頼結果を受付 AP へ返却する。
  - (4) WorkListener はワークマネージャが行う作業完了・受付拒否を監視しており、実行結果に対する処理を DB へ書き込む。
    ※ 受付拒否に関しては各ベンダーの製品に依存。

- ワークマネージャのキューイング（WebLogic の例）
  - ➢ クライアントから送られてきたジョブ依頼情報を直接キューに詰め込まず、ワークマネージャを介して、ジョブ依頼情報を制御する。
    （スレッドの割り当てもワークマネージャが行う）

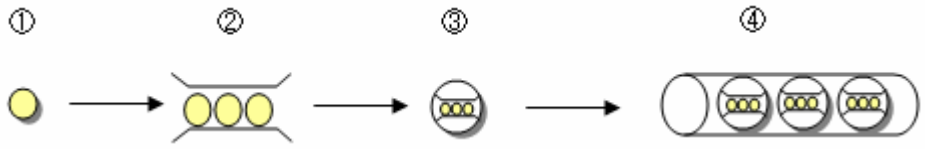


- AP サーバ上での処理フロー
  - AP サーバのワークマネージャを使用して、スケジュール毎にスレッドが割り当てられる。
  - 以下に、ジョブ依頼情報を受付けた際の AP サーバ上での動きを記述する。



(1) ジョブ依頼情報受付がジョブ依頼情報を渡して、JobStarter を実行する。
(2) JobStarterImpl が WorkManager の schedule()メソッドで JobExecutor のスケジュールを行う。
同時にスレッドを割り当てる。
(3) WorkManager が run()メソッドを起動し、JobExecutor が呼び出される。
(4) JobExecutor の work()メソッドを起動する事によって、ジョブが起動する。
(5) チャンクキューが作成される。
(6) (7) JobWorker をジョブ実行用スレッドに割り当てる。
(8) ワークマネージャが JobWorker を起動する。
(9) Collector によって入力データをチャンク単位でまとめる。
(10) まとめたチャンクをチャンクキューへ詰め込む。
(11) JobWorker がチャンクキューからチャンクを取得する。
(12) 処理対象のビジネスロジックを実行する。

- 処理フローにおける使用するキューのイメージ
  - 以下にキューに詰め込むまでのフローを記述する。



①Collector によって取得したデータをチャンクにまとめ、②チャンクキューに詰め込み、③チャンクキューを処理するキュープロセッサを Work でラップし、④ワークマネージャキューへ詰め込む。

- WorkListener（JobExecutorListener）の定義
  - WorkListener は作業完了・受付拒否を監視しており、WorkListener を拡張したジョブ実行結果ハンドラを用いて、実行結果に対する処理を DB へ書き込むことができる。
  - 作業が完了した場合は終了コードを返却し、受付が拒否された場合は、終了コードとして「-1」を返却する。
  - Commonj 対応機能を使用する際には『WorkManagerTaskContext-batch.xml』の設定が有効になるため『DefaultValueBean.xml』の設定は無効になる。

- ジョブ実行結果ハンドラ
  - WorkListener が監視していたジョブの実行結果を用いて、以下の実装を定義する事ができる。

項番	ハンドリング機能	実装クラス
1	実行結果をジョブ結果テーブルへ登録する。	jp.terasoluna.fw.batch.commonj.transaction.JobResultInfoHandlerImpl

- ジョブ結果テーブル内容
  - ジョブの処理結果を格納する。
  - 結果を確認するためだけのテーブルであり、ジョブ依頼番号がない場合でも対応できる。

項番	属性名	カラム名	必須	概要
1	ジョブ依頼番号	REQUEST_NO		ジョブ依頼連番。
2	ジョブID	JOB_ID		実行対象のジョブ ID。(bean 名)
3	起動状況	JOB_STATE		0:起動前、1:起動中、2:再起動中、3:正常終了、4:異常終了、7:中断/強制終了 バッチ登録時は 0 を初期値として登録する必要がある。
4	ジョブ終了コード	END_CODE		ジョブ処理終了後、返却される終了コード。
5	登録時刻	REGISTER_TIME		登録時刻

※ ジョブ結果テーブルのテーブル名及びカラム名は各プロジェクトで自由に変更できる。

■ 使用方法

◆ コーディングポイント

- JobStarter の実装を定義

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
…>
    <!-- プレースホルダ -->
    <import resource="classpath:common/PlaceHolderConfig.xml"/>

    <!-- ====================== ジョブStarter ====================== -->
    <bean id="jobStarter"
        class="jp.terasoluna.fw.batch.commonj.init.JobStarterImpl">
        <property name="workManager" ref="manager"/>
        <property name="workListener" ref="listener"/>
    </bean>

    <!-- ====================== ワークマネージャの設定 ====================== -->
    <bean id="manager"
        class="org.springframework.scheduling.commonj.WorkManagerTaskExecutor">
        <property name="workManagerName" value="${workManagerName}"/>
        <property name="resourceRef" value="true"/>
    </bean>

    <!-- ====================== ワークリスナー設定 ====================== -->
    <bean id="listener"
        class="jp.terasoluna.fw.batch.commonj.listener.JobExecutorListener">
        <property name="jobResultInfoHandler" ref="jobResultInfoHandler" />
    </bean>

    <!-- ====================== ジョブ結果用ハンドラ（JOB_RESULTテーブル） ================ -->
    <bean id="jobResultInfoHandler"
        class="jp.terasoluna.fw.batch.commonj.transaction.JobResultInfoHandlerImpl">
        <property name="updateDAO" ref="updateDAO" />
        <property name="transactionManager" ref="transactionManager" />
    </bean>
…
```

jobStarter の実装を設定

プロパティファイルで名前の設定

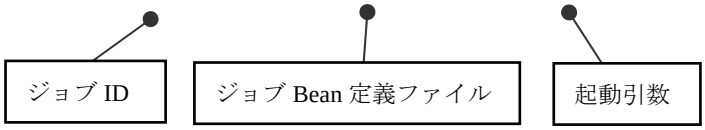
workListener の実装を設定

ジョブ実行結果ハンドラを指定

ジョブ実行結果ハンドラを設定

- パラメータの設定
  - JobStarter の実装に際し、ジョブを実行する場合は以下の３種類のパラメータを設定する必要がある。

```
public class StartBatchBLogicImpl implements BLogic<Map<String, Object>> {
…
        String[] args = {"Sample","2007","0"};
        //ジョブの起動
        int jobEndCode = jobStarter.execute("Job01", "sample/JOB01.xml", args);
…
}
```



- チャンクキューの設定（workQueueFactory.properties）
  - キューの設定をプロパティファイルで切り替える事が可能。
    （通常のバッチ用キューに対しては、コメントアウト等で対応）

```
workQueueFactory.class=jp.terasoluna.fw.batch.commonj.usequeue.WorkManagerTaskWorkQueueFactory
```

- ジョブ Bean 定義ファイルの設定例（WorkManagerTaskContext-batch.xml）

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
         ... >
   <!-- ==================== WorkExecutor定義 ==================== -->
   <bean id="workerExecutorService"
        class="jp.terasoluna.fw.batch.commonj.WorkManagerExecutorService">
      <property name="defaultJobExceptionHandler" ref="defaultJobExceptionHandler" />
      <property name="exceptionHandlerMap" ref="exceptionHandlerMap" />
      <property name="workManager" ref="taskExecutor"/>
      <property name="workMapListener" ref="workListener"/>
   </bean>
   <bean id="taskExecutor"
        class="org.springframework.scheduling.commonj.WorkManagerTaskExecutor">
      <property name="workManagerName" value="${workManagerName}"/>
      <property name="resourceRef" value="true"/>
      <property name="jndiTemplate" ref="jndiTemplateForWorkManager"/>
   </bean>
   <bean id="workListener"
        class="jp.terasoluna.fw.batch.commonj.listener.WorkQueueListener"/>
   <!-- ==================== 監視(File)定義 ==================== -->
   <bean id="scheduledTask"
        class="org.springframework.scheduling.commonj.ScheduledTimerListener">
      <!-- 8秒ごとにセット -->
      <property name="delay" value="0"/>
      <property name="period" value="8000"/>
      <property name="fixedRate" value="false"/>
      <property name="runnable" ref="endFileChecker"/>
   </bean>
   <bean id="timerFactory"
        class="org.springframework.scheduling.commonj.TimerManagerFactoryBean">
      <property name="timerManagerName" value="${timerManagerName}"/>
      <property name="resourceRef" value="true"/>
      <property name="shared" value="true"/>
      <property name="scheduledTimerListeners">
         <list>
            <ref bean="scheduledTask" />
         </list>
      </property>
      <property name="jndiTemplate" ref="jndiTemplateForTimerManager"/>
   </bean>
</beans>
```

JNDI に対するサーバ依存の設定が可能

- web.xml の設定例

```
<?xml version="1.0" encoding="UTF-8" ?>

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
…
        <resource-ref>
                <res-ref-name>wm/BatchWorkManager</res-ref-name>
                <res-type>commonj.work.WorkManager</res-type>
                <res-auth>Container</res-auth>
                <res-sharing-scope>Shareable</res-sharing-scope>
        </resource-ref>

        <resource-ref>
                <res-ref-name>timer/FileCheckTimer</res-ref-name>
                <res-type>commonj.timers.TimerManager</res-type>
                <res-auth>Container</res-auth>
                <res-sharing-scope>Unshareable</res-sharing-scope>
        </resource-ref>

</web-app>
```

WorkManager の設定

TimerManager の設定

◆ 拡張ポイント

- なし

■ 関連機能

- 『BE-01 同期型ジョブ起動機能』
- 『BE-02 非同期型ジョブ起動機能』
- 『BE-03 ジョブ実行管理機能』

■ 使用例

- 簡易サンプルプロジェクト(AP サーバ対応) (sample-batch_onWebAPServer)

■ 備考

- なし。