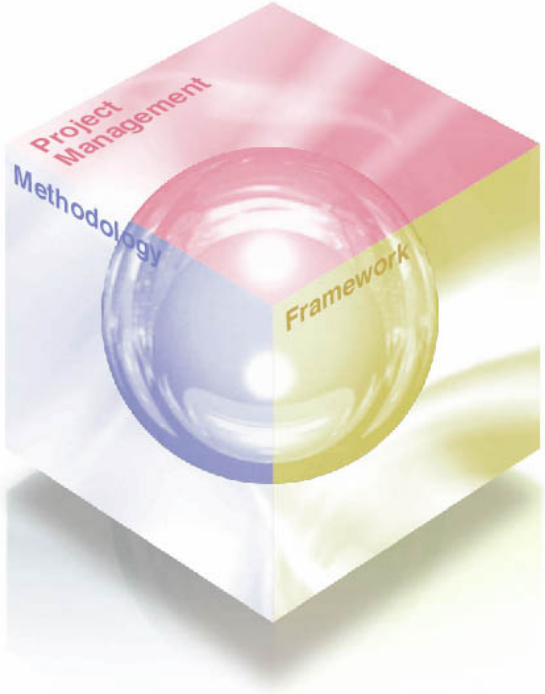
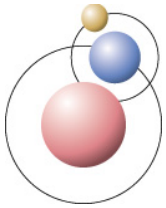
TERASOLUNA® Batch Framework for Java
アーキテクチャ説明書



株式会社NTTデータ

TERASOLUNA®
BEST SOLUTION FOR SYSTEM DEVELOPMENT

NTT DATA



- 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。

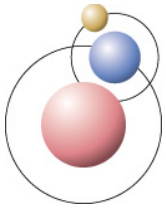
1. 本ドキュメントの著作権及びその他一切の権利は、NTTデータあるいはNTTデータに権利を許諾する第三者に帰属します。
2. 本ドキュメントの一部または全部を、自らが使用する目的において、複製、翻訳、翻案することができます。ただし本ページの規約全文、およびNTTデータの著作権表示を削除することはできません。
3. 本ドキュメントの一部または全部を、自らが使用する目的において改変したり、本ドキュメントを用いた二次的著作物を作成することができます。ただし、「TERASOLUNA® Batch Framework for Java　アーキテクチャ説明書」あるいは同等の表現を、作成したドキュメント及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。

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

目次

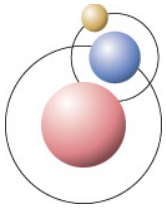
- はじめに
- 第一章 アーキテクチャ概観
- 第二章 フレームワーク機能説明
  - BA-01 トランザクション管理機能
  - BB-01 データベースアクセス機能
  - BC-01 ファイルアクセス機能
  - BC-02 ファイル操作機能
  - BD-01 ビジネスロジック実行機能
  - BD-02 対象データ取得機能
  - BD-03 コントロールブレイク機能
  - BE-01 同期型ジョブ起動機能
  - BE-02 非同期型ジョブ起動機能
  - BE-03 ジョブ実行管理機能（一部非推奨機能）
  - BE-04 リスタート機能
  - BE-05 処理結果ハンドリング機能
  - BF-01 メッセージ管理機能
  - BH-01 例外ハンドリング機能
  - BI-01 Commonj対応機能（非推奨機能）
- 第三章 拡張ポイント
- 第四章 用語説明



はじめに

本資料では、
「TERASOLUNA Batch Framework for Java」（以下：TERASOLUNA-Batch）
ver2.0.2.0のアーキテクチャを説明する。
本資料が想定している読者はソフトウェアアーキテクトである。ソフトウェア
アーキテクトが、TERASOLUNA-Batchの概要としくみを把握することを目的とする。

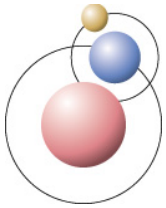
- 「第一章 アーキテクチャ概観」について
  - TERASOLUNA-Batchの基本的なクラス構成など、アーキテクチャについて概観を説明する。
- 「第二章 フレームワーク機能説明」について
  - TERASOLUNA-Batchの各機能について、概要やコーディングポイントを解説する。
  - 詳細な機能説明については『機能説明書』を参照のこと。
- 「第三章 拡張ポイント説明」について
  - TERASOLUNA-Batchをプロジェクトへ適用する際に、プロジェクトの要件に合わせて拡張できるポイントを解説する。
  - 詳細な拡張ポイントの説明については『機能説明書』を参照のこと。
- 「第四章 用語説明」について
  - TERASOLUNA-Batchの各種ドキュメントを参照する際に、前提知識とすべき用語を解説している。



はじめに

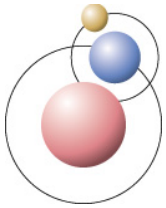
依存するオープンソースライブラリー覧

オープンソースライブラリ名	バージョン	TERASOLUNAモジュール(terasoluna-*)					
		batch-core	commons	dao	filedao	ibatis	validator
cglib-nodep.jar	2.1_3		○	○		○	
commonj-twm.jar	1.1	○					
commons-beanutils.jar	1.7	○	○		○		○
commons-collections.jar	3.2	○			○		
commons-dbcp.jar	1.2.2	○				○	
commons-digester.jar	1.8	○					○
commons-jxpath.jar	1.2		○				○
commons-lang	2.3	○	○		○		○
commons-logging.jar	1.1	○	○	○	○	○	○
commons-pool.jar	1.3	○				○	
commons-validator.jar	1.3.1	○					○
easymock.jar	2.3			○(テスト用)			
ibatis.jar	2.3.0.677	○				○	
jakarta-oro.jar	2.0.8						○
junit.jar	3.8.2	○(テスト用)	○(テスト用)	○(テスト用)	○(テスト用)	○(テスト用)	○(テスト用)
junit-addons.jar	1.4	○(テスト用)	○(テスト用)	○(テスト用)	○(テスト用)	○(テスト用)	○(テスト用)
mockrunner-jdbc.jar	0.3.7		○(テスト用)				
servlet-api.jar	2.4		○	○			
spring.jar	2.0.6	○	○	○		○	○
spring-mock.jar	2.0.6						○(テスト用)
spring-modules-validation.jar	0.8	○					○

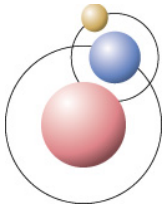


はじめに

- 動作確認環境
  - 対応JDK
    - J2SE5.0/6.0
  - 対応データベース
    - Oracle 11G (11.1.0)
    - Oracle 10G (10.2.0)
    - Oracle 9i (9.2.0)
    - PostgreSQL8.2
  - 対応WebAPサーバ
    - WebLogic Server 9.2



第一章 アーキテクチャ概観

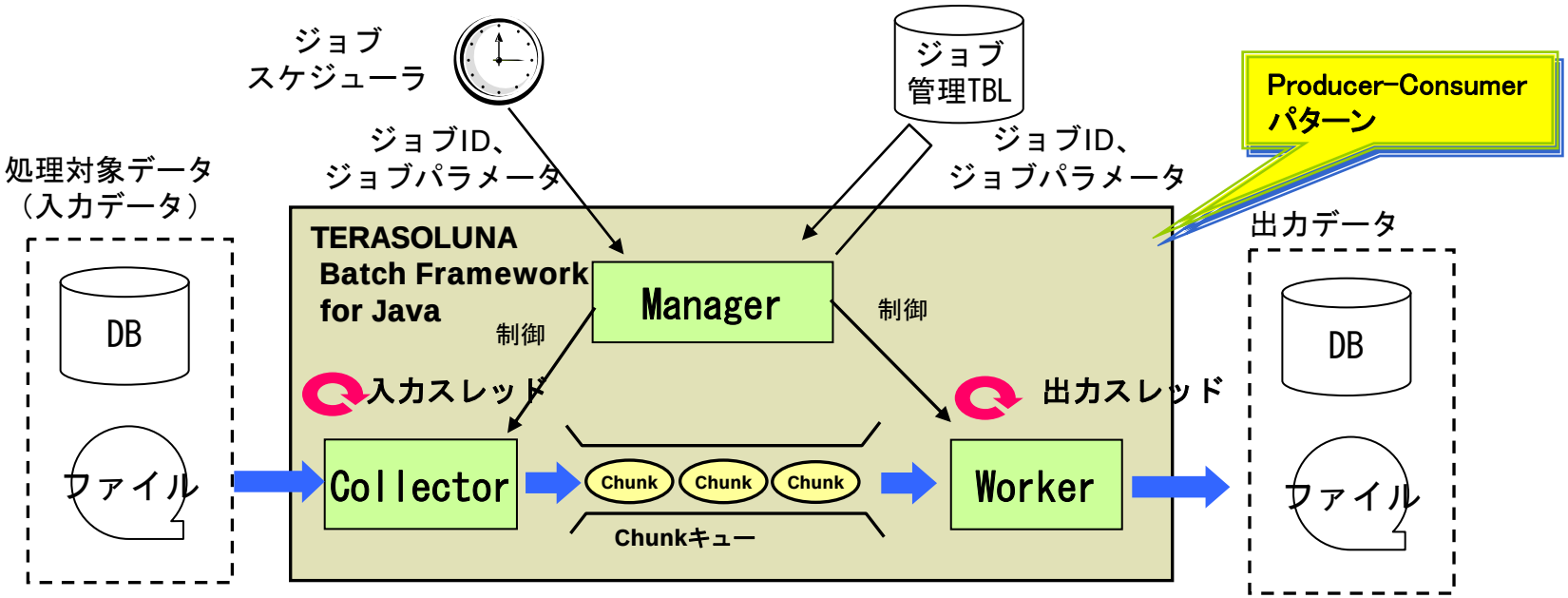


アーキテクチャの設計思想

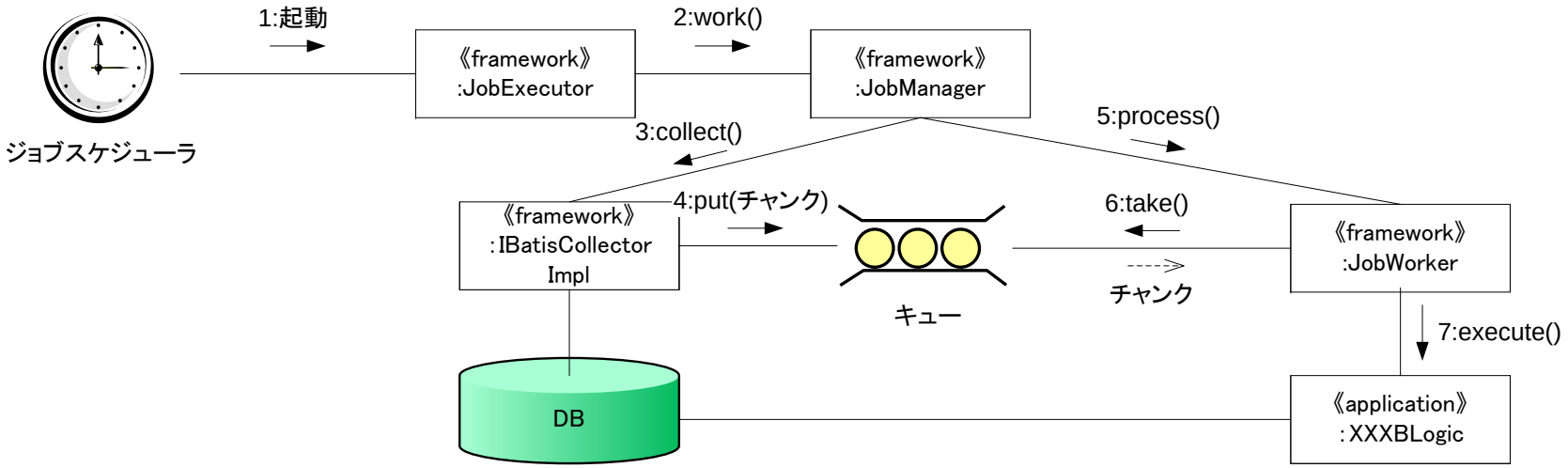
- 入力処理と出力処理の分離
  - 入力処理と出力処理を分離することで、アプリケーションロジックのループ制御やトランザクション管理の標準化を実現
  - 「アーキテクチャ概要」、「基本的な処理の流れ」、「スレッドの構成」を参照
- 柔軟な階層構造
  - 柔軟な階層構造により、様々な処理形態や起動形態に対応
  - 「分割キーによる多重化」、「常駐プロセスでの非同期実行」を参照
- オブジェクト生成及び組立ての一元化
  - オブジェクト生成及び組立てをDIコンテナ（Spring Framework）で行うことで、オブジェクトの組み合わせを柔軟に変更可能
  - 設定ファイル作成の煩雑さは、Bean定義雛形ファイルを提供することで、軽減
  - 「設定ファイルの構成」、「設定ファイルの内部構造」、「設定ファイルの具体例」を参照

アーキテクチャ概要

- 入力処理と出力処理のモジュール、スレッドを分割する。
  - 入力、出力でのI/O待ちが発生しても、バッチアプリケーションでは待ちを発生させない
- 入力処理と出力処理の間は、キューによりデータの受け渡しを行う。
  - キューイングするデータは、ビジネスロジックの入力データを一定個数分まとめたChunk
  - Chunkは、入出力間のキャッシュとなり、またトランザクション単位となる



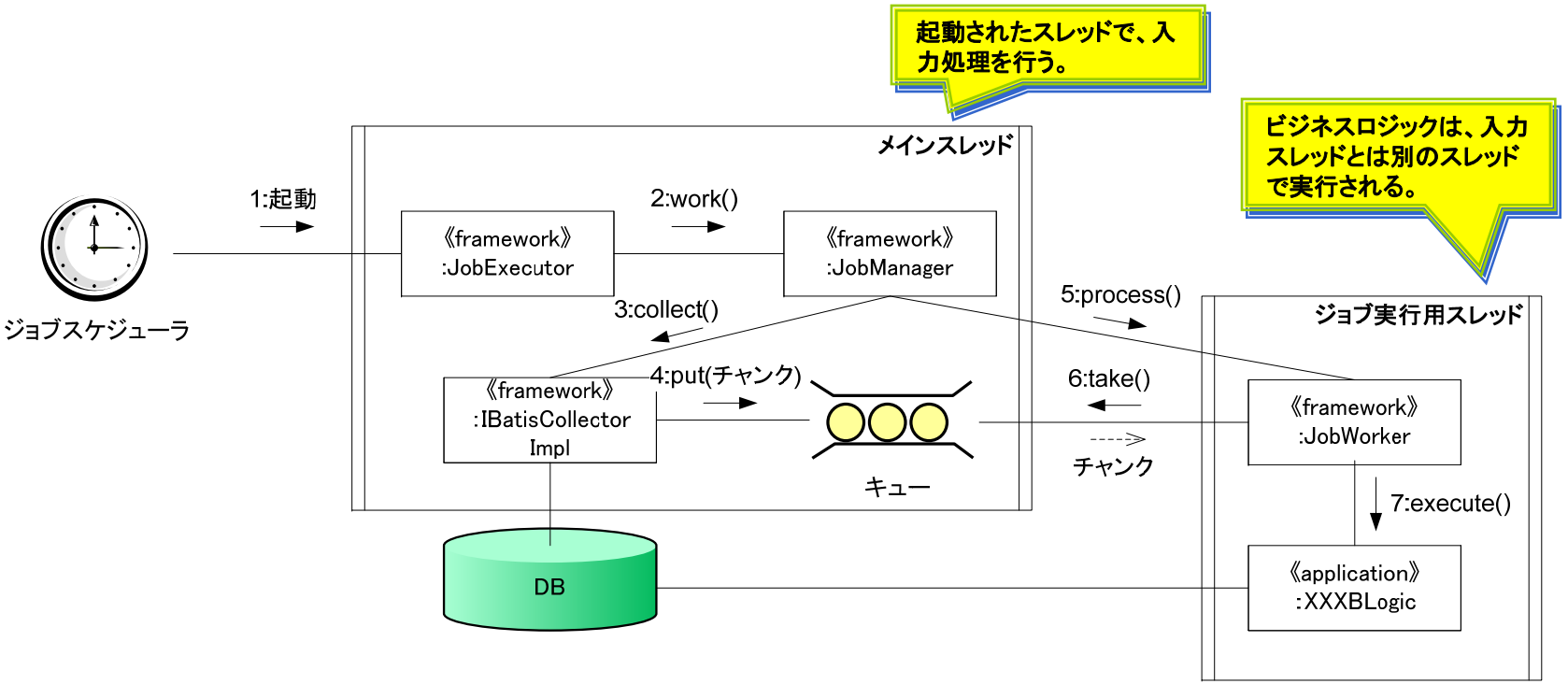
基本的な処理の流れ



1. ジョブスケジューラがJobExecutorを起動する。JobExecutorはパラメータを解析し、ジョブBean定義ファイルを読込む。
2. JobExecutorがJobManagerを起動する。
3. JobManagerがIBatisCollectorImplを起動し、DBよりバッチ入力データを取得する。
4. IBatisCollectorImplがバッチ入力データを物理トランザクションの単位ごとにチャンクとして纏めてキューに登録する。
5. JobManagerがスレッドプールよりジョブ実行用スレッドを取得し、JobWorkerを起動する。
6. JobWorkerがキューからチャンクを取得する。
7. JobWorkerがチャンクからバッチ入力データのリストを取得し、各バッチ入力データについてビジネスロジックを実行する。

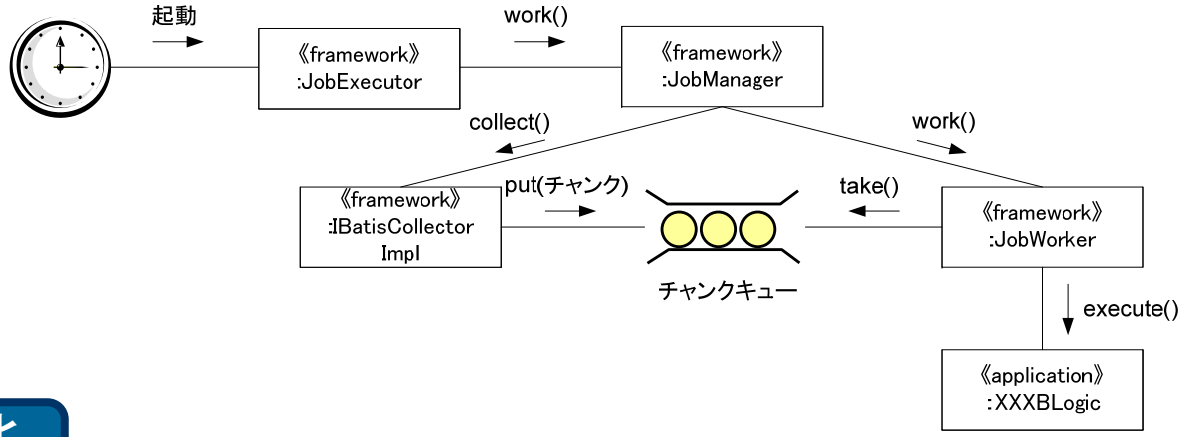
スレッドの構成

- 入力処理と出力処理のモジュール、スレッドは分割する。
  - 起動されたスレッドで入力処理を行う。
  - ビジネスロジックは、入力スレッドとは別のスレッドで実行される。

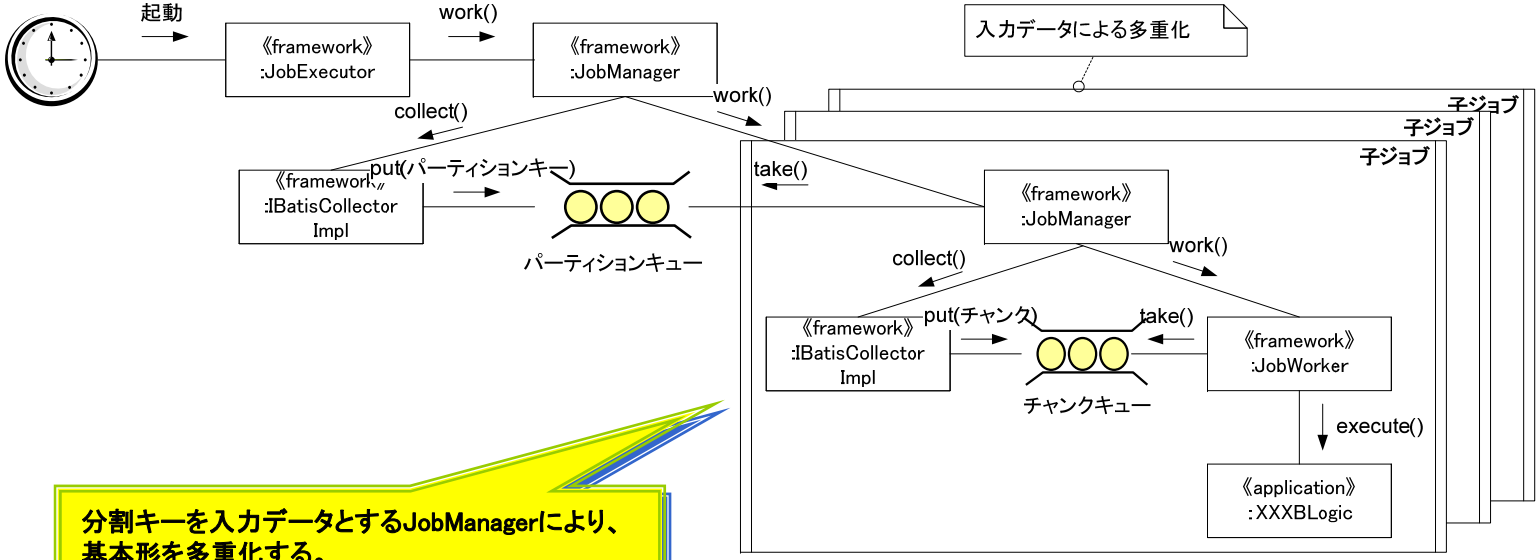


分割キーによる多重化

基本形（多重化なし）



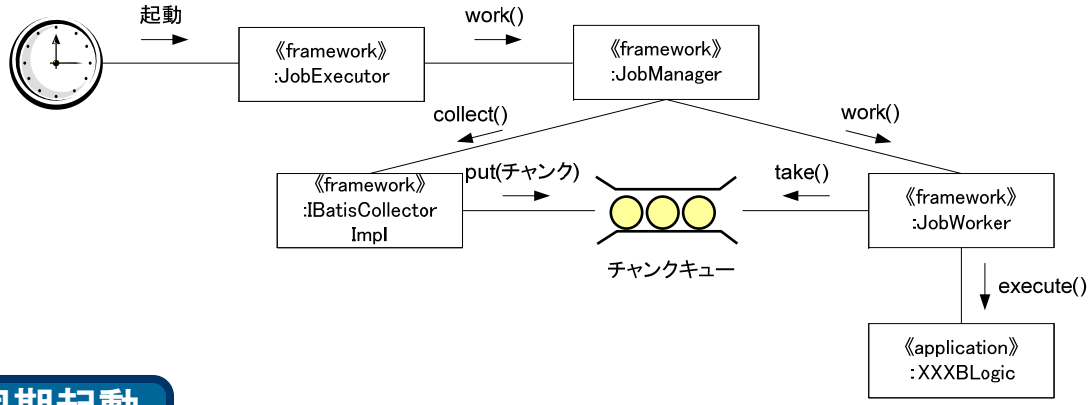
分割キーによる多重化



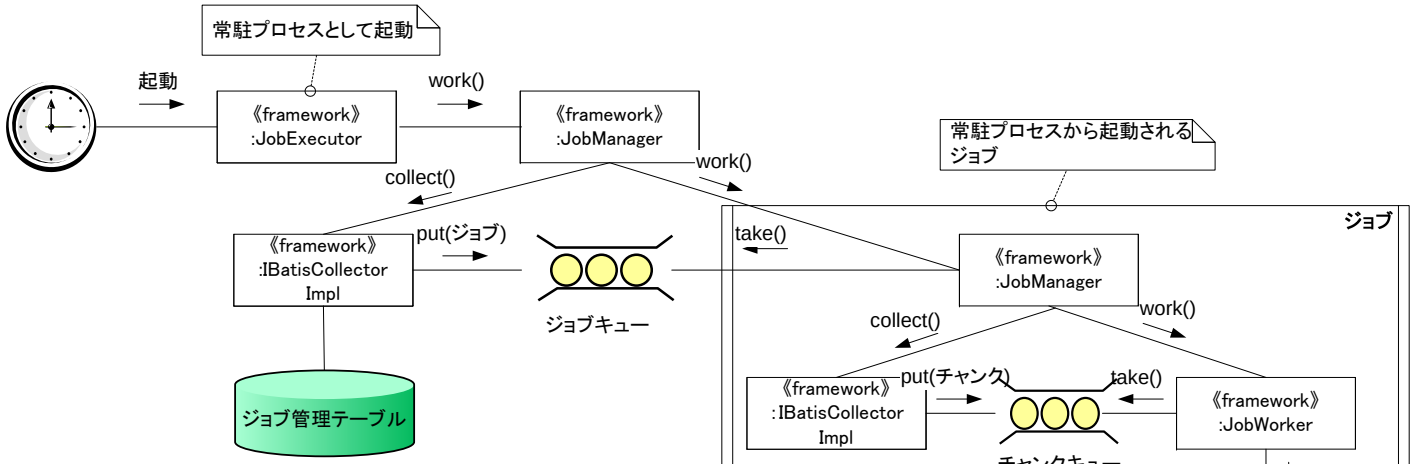
分割キーを入力データとするJobManagerにより、基本形を多重化する。

常駐プロセスでの非同期起動

基本形（同期型起動）



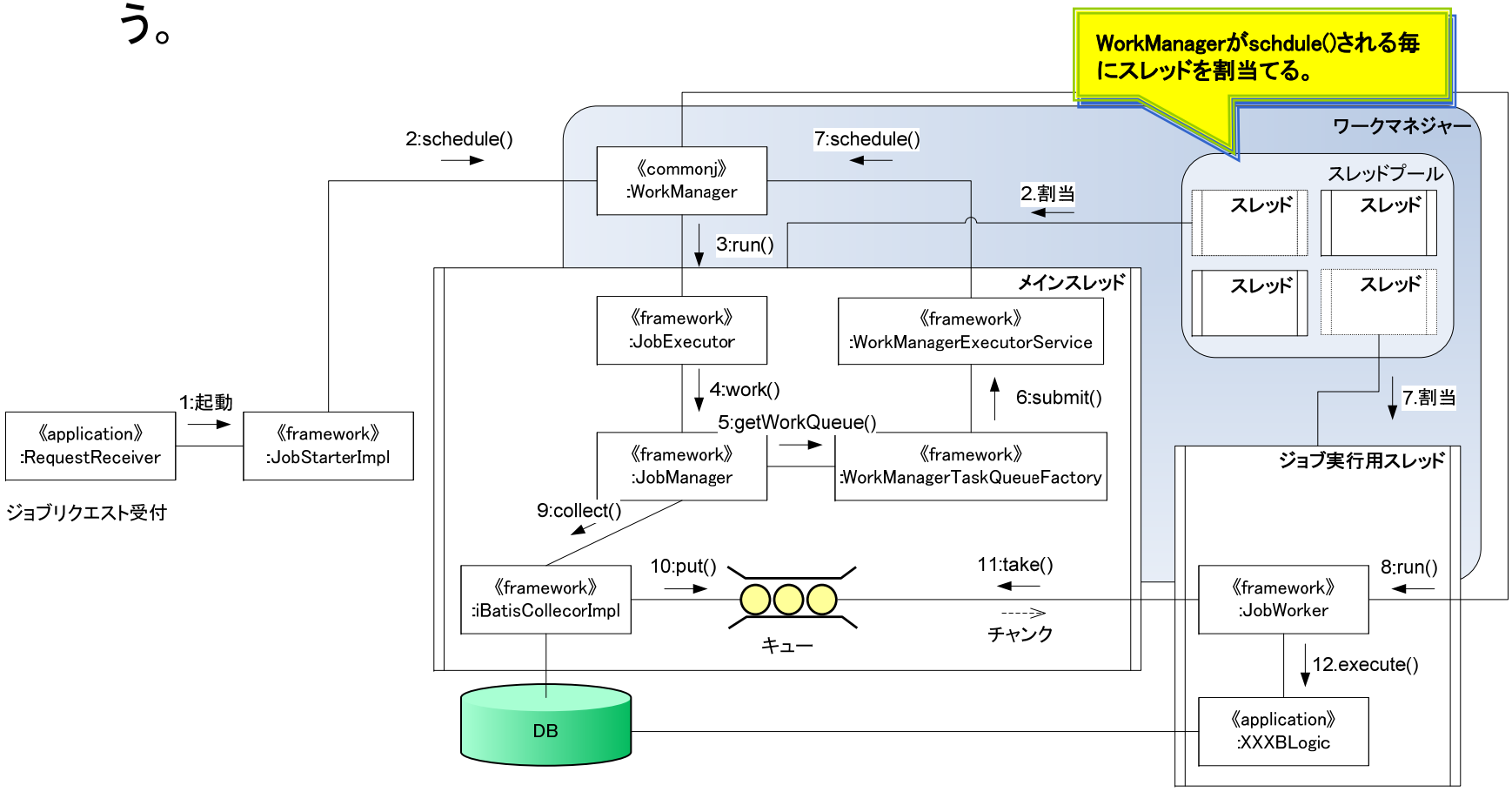
常駐プロセスでの非同期起動



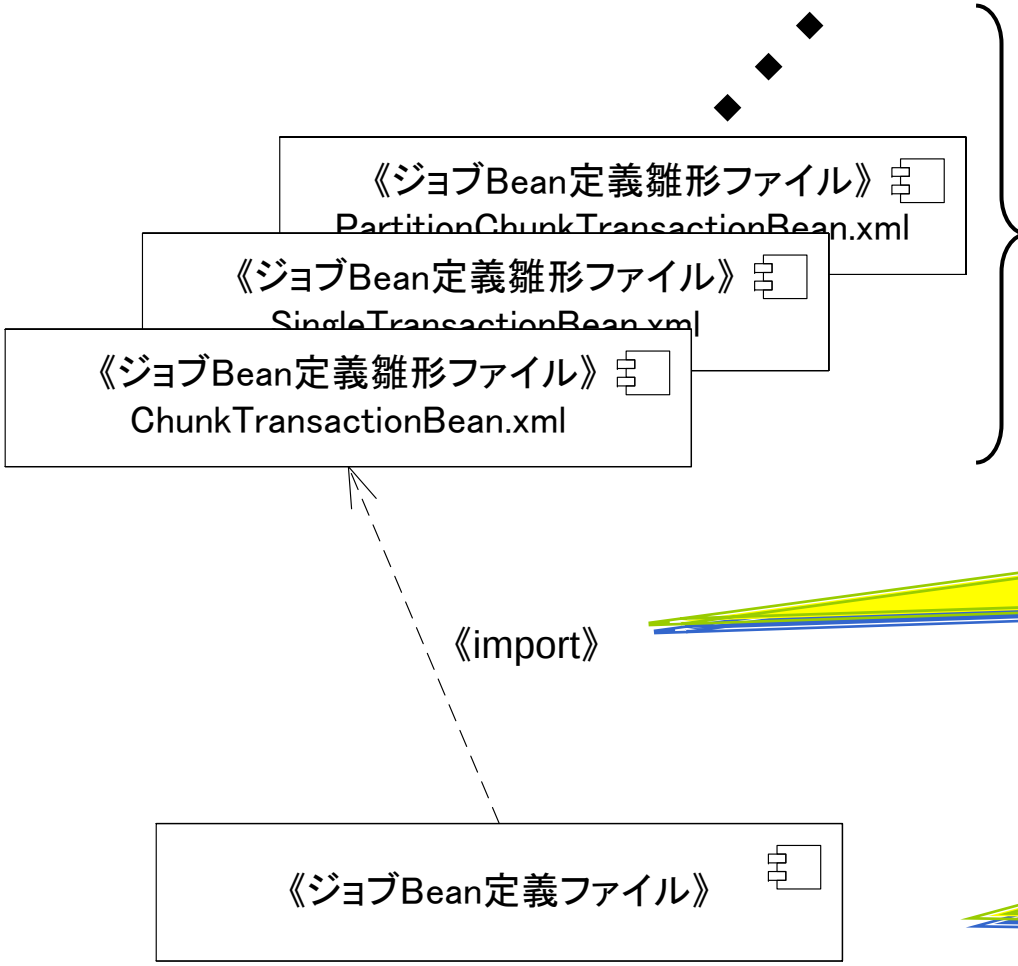
常駐化されたJobManagerにより、基本形ジョブを起動する。

アプリケーションサーバ上の非同期起動

- アプリケーションサーバのワークマネジャーを利用しスレッド管理を行う。



設定ファイルの構成

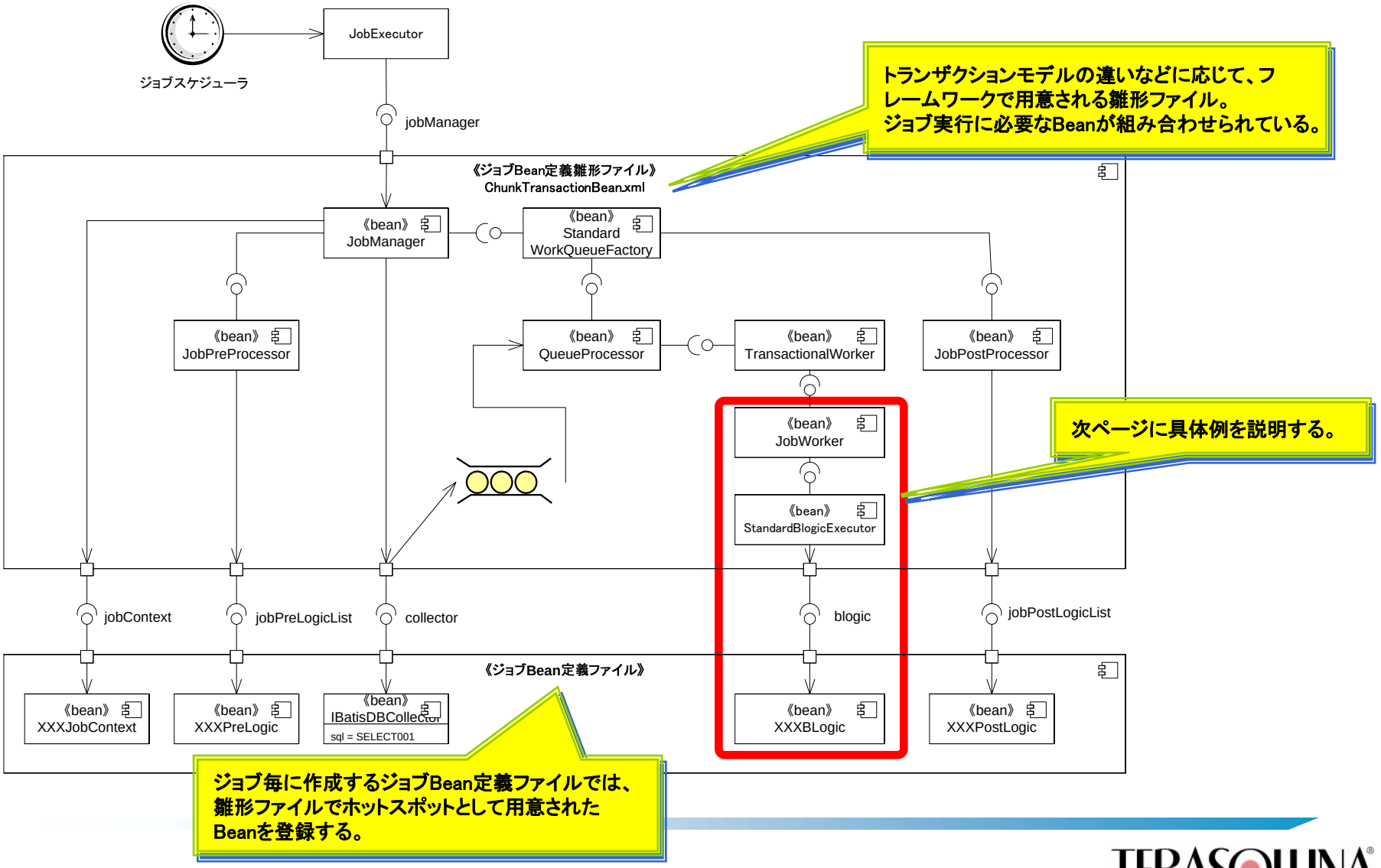


《ジョブBean定義雛形ファイル》
処理モデル毎に作成するBean定義ファイル。
キー分割による多重化の有無やトランザクションモデルの種類に応じた情報が定義される。
必要に応じてアーキテクトが本ファイルを編集もしくは追加する。

ジョブBean定義ファイルでは、個々の処理モデルに応じたジョブBean雛形ファイルをimportする。

《ジョブBean定義ファイル》
各ジョブに特化した設定を記載するBean定義ファイル。

設定ファイルの内部構造



設定ファイルの具体例

ジョブBean定義雛形ファイル

```
…
<bean id=“jobWorker“  class=“jp.terasoluna.…JobWorker“ >
   <property name=“blogicExecutor“ ref=“blogicExecutor“ />
     …
</bean>

<bean id=“blogicExecutor“ class=“jp.terasoluna.…StandardBLogicExecutor”>
   <property name=“blogic“ ref=“blogic“ />
    …
</bean>
…
```

雛形ファイルでは、処理モデル毎に必要なBeanが組み合わせられている。

雛形ファイルでは、ジョブ毎に異なるBeanは参照だけが登録されている。

ジョブBean定義ファイル

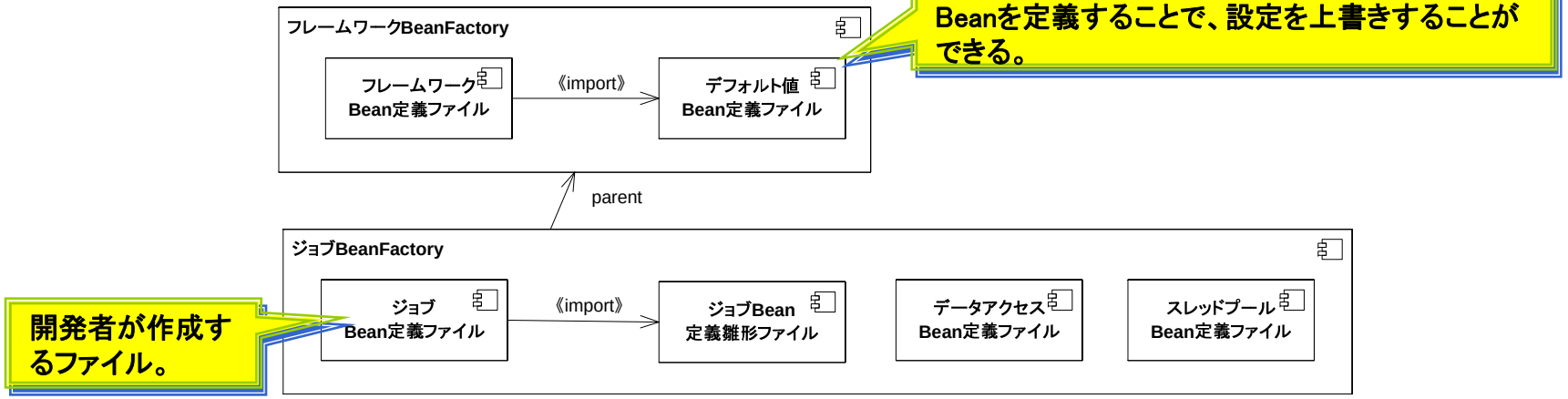
```
<import  resource=“../chunkTransactionBean.xml“ />


<bean id=“blogic“ class=“jp.terasoluna.….XXXBLogic“>
   <property name=“queryDAO“ ref=“queryDAO“ />
    ….
</bean>
….
```

ジョブBean定義ファイルでは、処理モデルに応じた雛形ファイルをimportする。

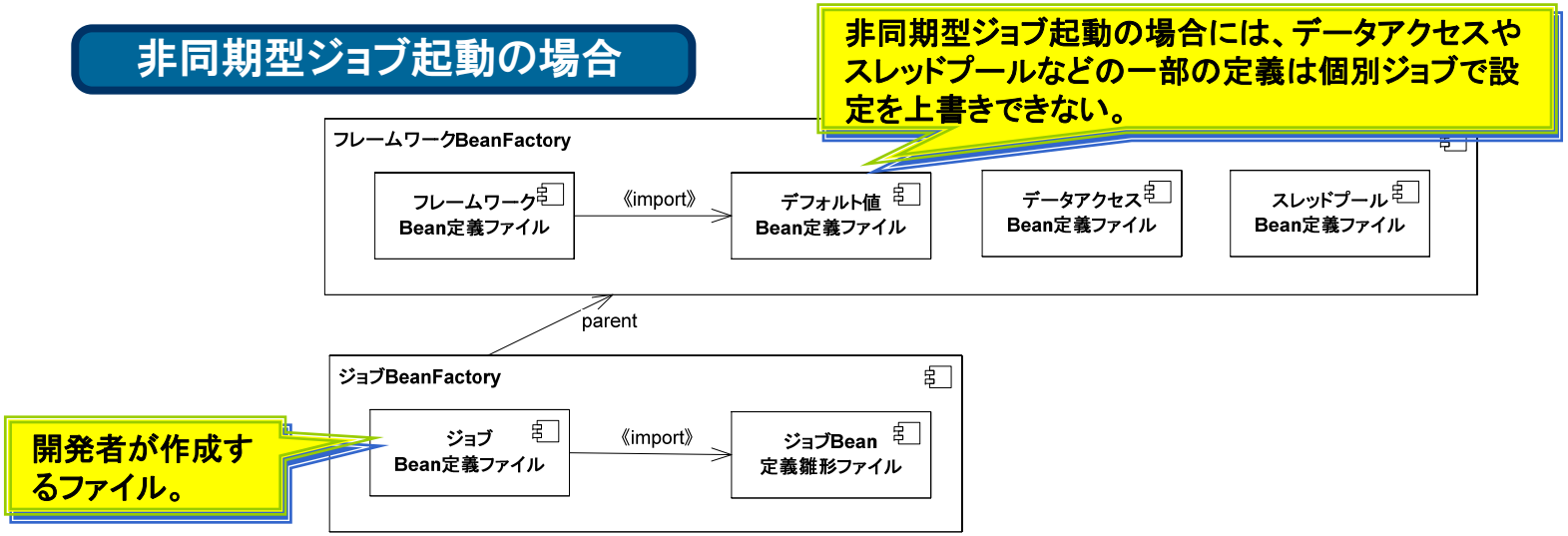
ジョブBean定義ファイルでは、ジョブ毎に異なる実装を用意する必要のあるBeanだけを登録する。

設定ファイル群と実行時BeanFactoryの関係

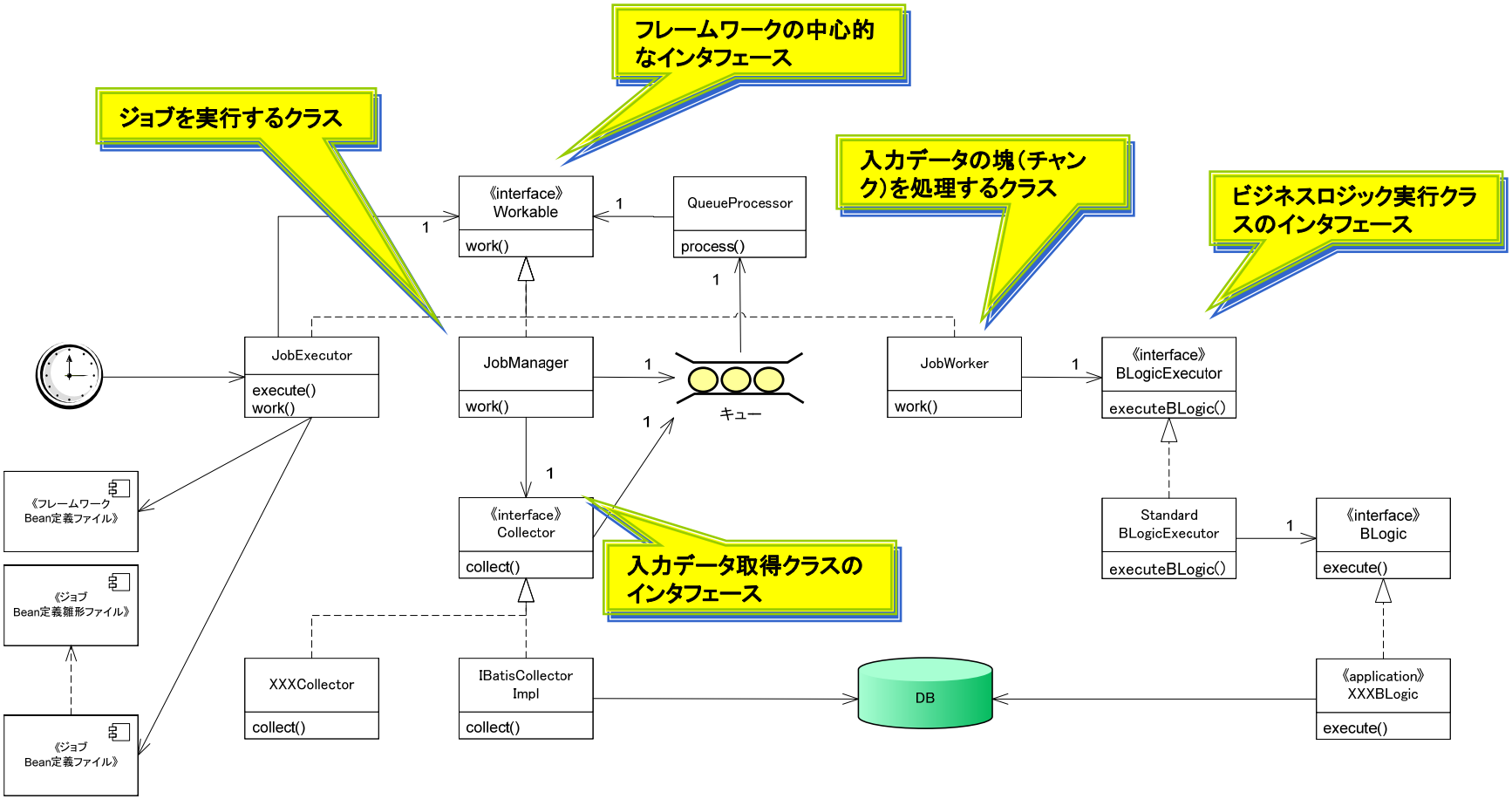
同期型ジョブ起動の場合



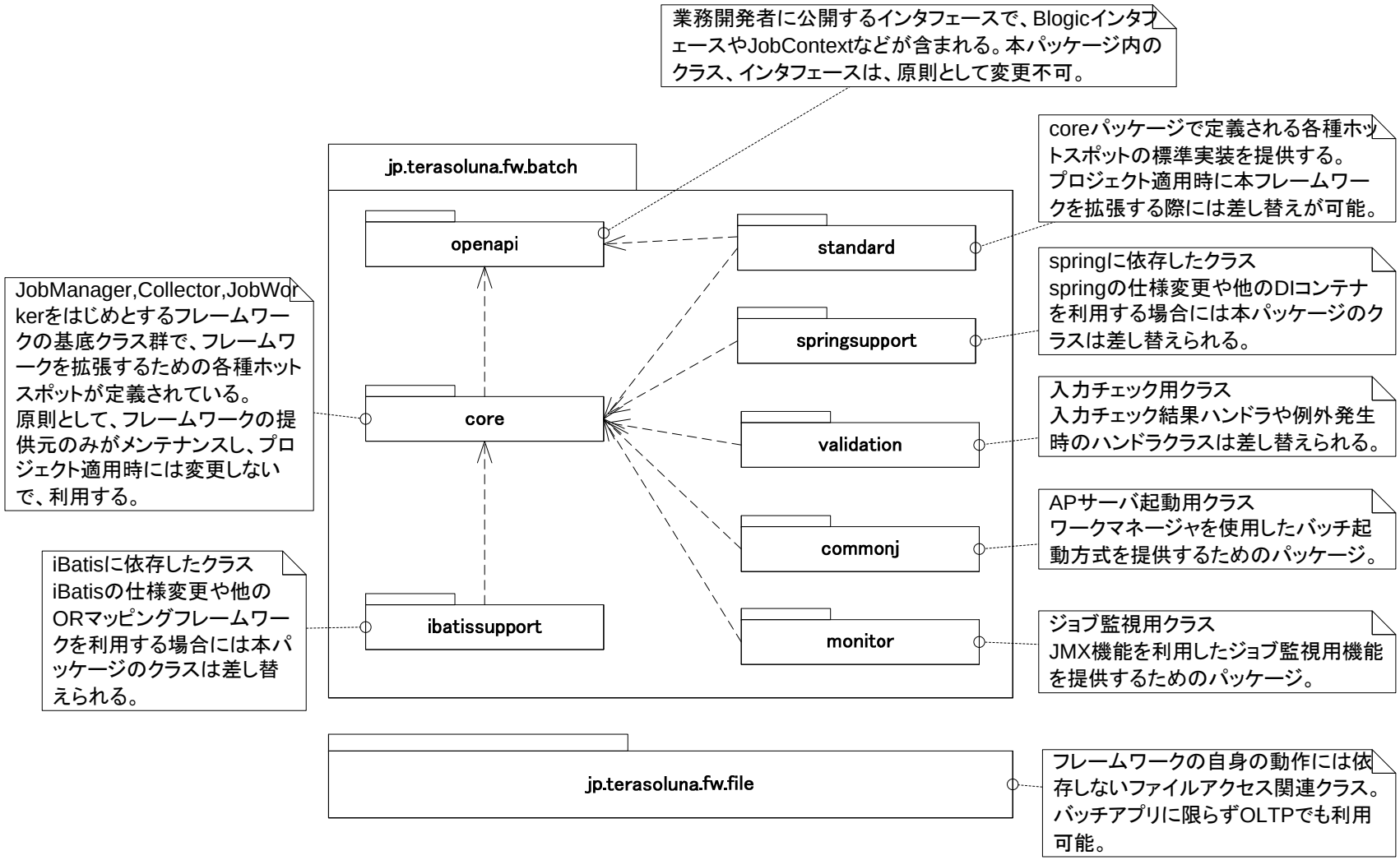
非同期型ジョブ起動の場合



基本的なクラス構造

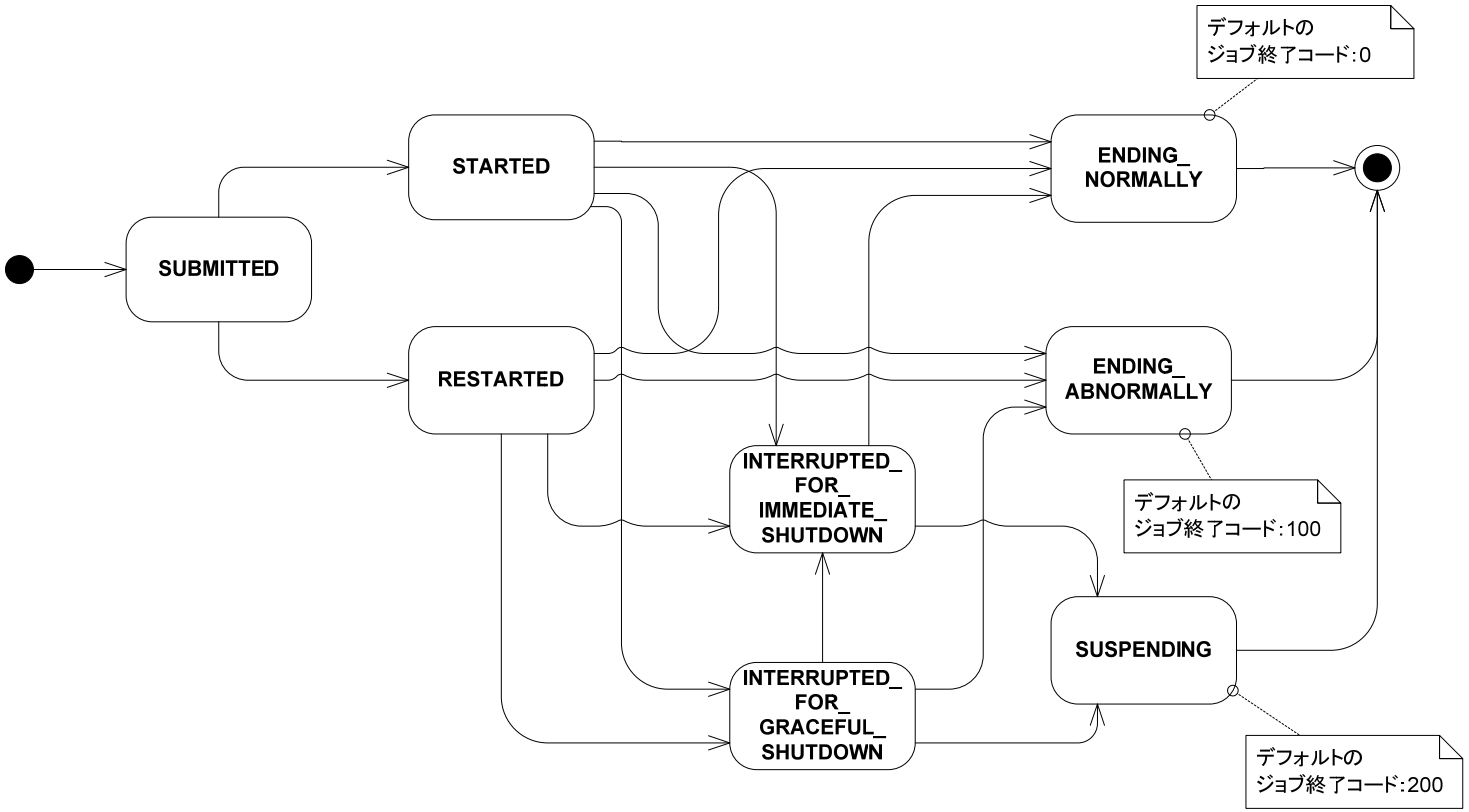


主要なパッケージの構成



ジョブの実行ステータスとジョブ終了コード

- ジョブ実行ステータスは、フレームワーク内部での実行制御に使用される。
- ビジネスロジック等でジョブ終了コードが指定されなかった場合には、デフォルトのジョブ終了コードが利用される。
  - デフォルトのジョブ終了コードは、DefaultValueBean.xmlを修正して変更することができる。



スレッドの設定①（同期型起動、通常ジョブ）

同期型起動用スレッドプール定義ファイル

ThreadPoolContext-batch.xml

```
…
<bean id="workerExecutorService" class="jp.terasoluna.fw.batch.....XXXExecutorService">
    <constructor-arg ref="threadSize"/>
</bean>
…
```

スレッドプールを管理するBean。

ジョブBean定義雛形ファイル

```
…
<bean id="…." >
  <property name="workerExecutorService" ref="workerExecutorService" />
    …
</bean>

<bean id="threadSize" class="java.lang.Integer">
  <constructor-arg value="1" />
</bean>
…
```

ジョブBean定義雛形ファイルで、その雛形のジョブに必要なスレッド数が指定されている。

ジョブBean定義ファイル

….（スレッドの設定なし）….

スレッドの設定②（同期型起動、分割ジョブ）

同期型起動用スレッドプール定義ファイル

ThreadPoolContext-batch.xml

```
...
<bean id="workerExecutorService" class="jp.terasoluna.fw.batch.....XXXExecutorService">
    <constructor-arg ref="threadSize"/>
</bean>
...
```

ジョブBean定義雛形ファイル

```
<bean id="...." >
    <property name="workerExecutorService" ref="workerExecutorService" />
    <property name="multiplicity" ref="multiplicity"/>
    ...
</bean>

<bean id="threadSize"
      class="jp.terasoluna.fw.batch.springsupport.standard.ThreadSizeFactoryBean">
  <property name="multiplicity" ref="multiplicity"/>
</bean>
```

ジョブBean定義雛形ファイルでは、ジョブの多重度に応じてスレッド数を決定する。

多重度数×2に合わせて、DBコネクションプール数を変更すること。

ジョブBean定義ファイル

```
<bean id="multiplicity" class="java.lang.Integer">
  <constructor-arg><value>1</value></constructor-arg>
</bean>
```

分割ジョブの場合には、ジョブBean定義ファイルで、ジョブの多重度を指定する。
multiplicityは、フレームワーク内部の多重化処理、およびスレッド数の設定で利用される。

スレッドの設定③（非同期型起動）

非同期型起動用スレッドプール定義ファイル

ThreadPoolContext-AsyncBatch.xml

```
…
<bean id="workerExecutorService"  class="jp.terasoluna.fw.batch.....XXXExecutorService">
      <constructor-arg value="20"/>
</bean>
…
```

スレッドプールを管理するBean。
非同期型起動の場合には、非同期バッチデーモン起動中には固定数のスレッドプールが使用される。
以下を考慮して、適切な値を設定する。
・各ジョブでCollectorとWorkerが別スレッドで実行される
・分割ジョブを使う場合には、子ジョブの多重度
・非同期起動するジョブの多重度

スレッド数に合わせて、DBコネクションプール数を変更すること。

ジョブBean定義雛形ファイル

```
…
<bean id="…." >
    <property name="workerExecutorService" ref="workerExecutorService" />
      …
</bean>

<bean id="threadSize" class="java.lang.Integer">
   <constructor-arg value="1" />
</bean>
…
```

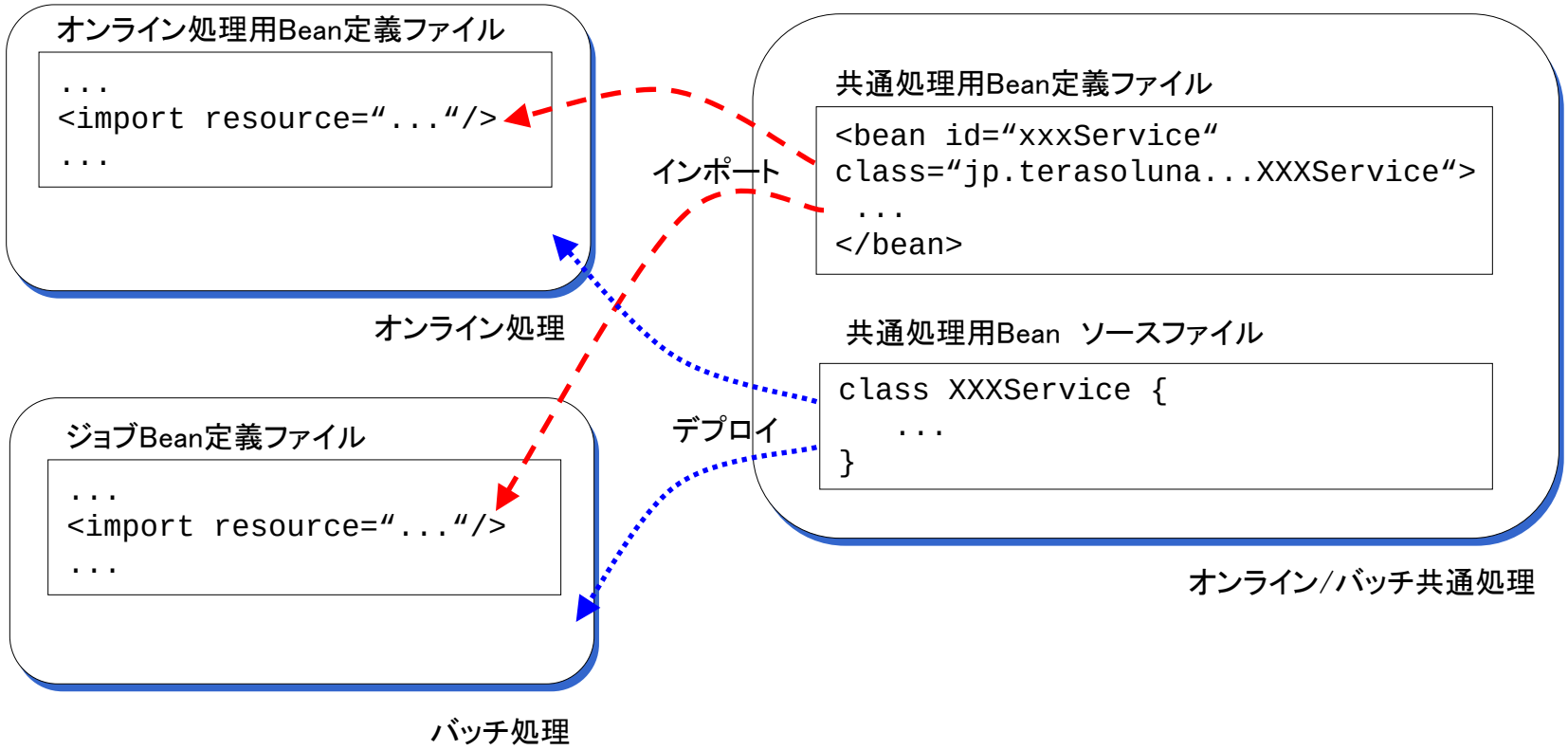
ジョブBean定義雛形ファイルで指定されているスレッドサイズは非同期型起動の場合には無視される。

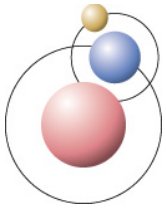
ジョブBean定義ファイル

….（スレッドの設定なし）….

オンライン処理とバッチ処理の共通化

- オンライン処理のフレームワークとしてTERASOUNA Server Framework for Java（Spring版）を用いている場合には、オンライン処理とバッチ処理でモジュールや設定を共通化することができる。

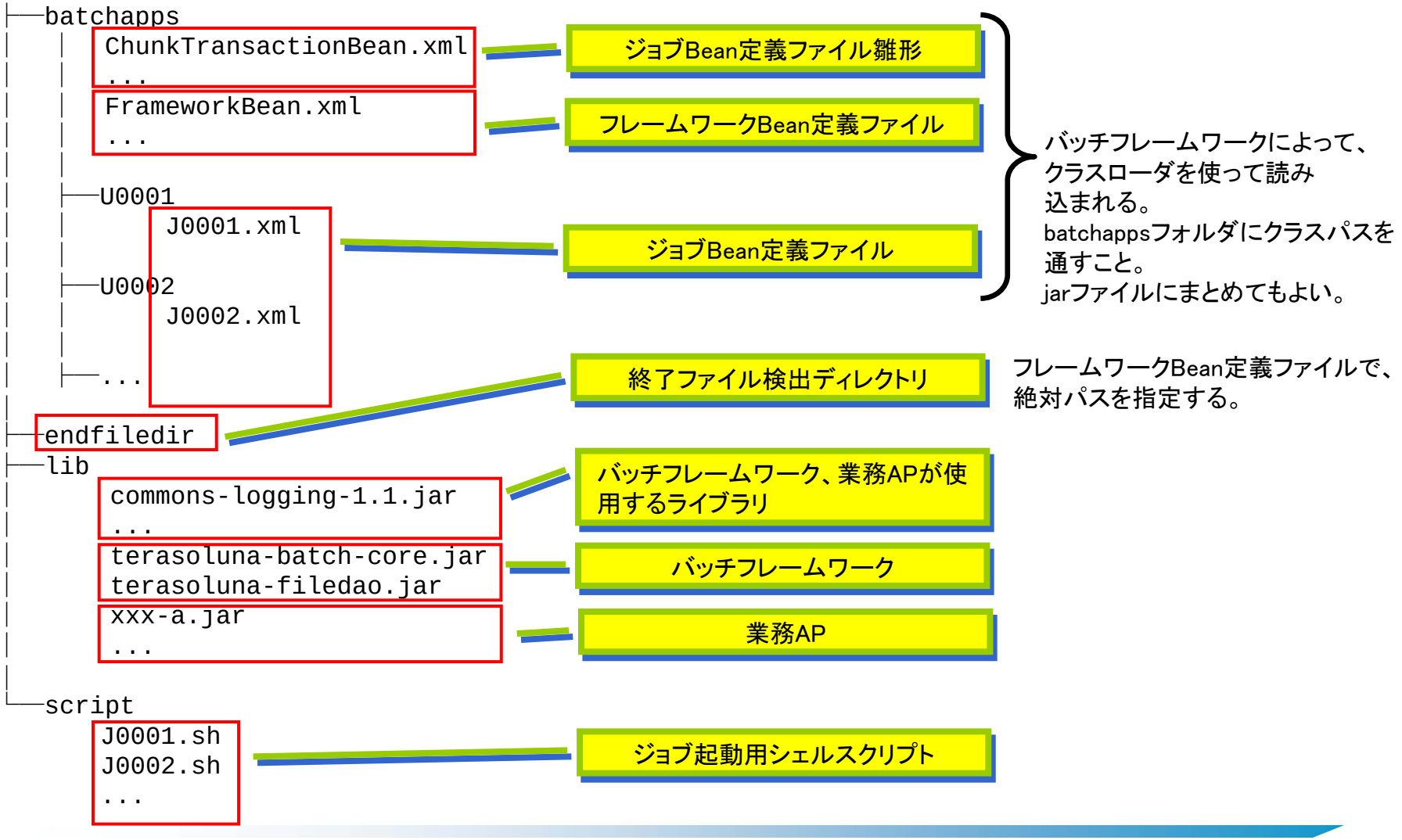


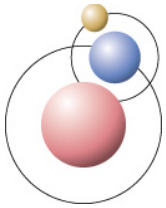


オンライン処理とバッチ処理の共通化での留意点

- トランザクションの設定
  - オンライン処理とバッチ処理とでは、一般にトランザクションの考え方が異なる。オンライン処理では1要求電文に対してACID属性を満たすようにトランザクションを設定する場合が多いのに対して、バッチ処理では処理効率向上のために複数データをまとめてトランザクションとして処理する場合が多い。
  - 単純に呼び出し元のトランザクションに参加する共通処理であれば、オンライン処理とバッチ処理での共通化を検討することができる。しかし、オンライン処理から呼ばれることを前提とした共通処理をバッチ処理から呼び出す場合には、トランザクション範囲に問題ないか留意すること。
- 入力チェック
  - バッチ処理でのファイルに対する入力チェックの定義は、TERASOUNA Server Framework for Java（Spring版）での入力チェックを共通化することができる。
- メッセージ取得処理
  - メッセージの管理方法は、TERASOUNA Server Framework for Java（Spring版）とTERASOLUNA-Batchで共通であり、統一された管理方法でメッセージを管理することができる。

実行時のフォルダ構成例





第二章 フレームワーク機能説明

BA-01 トランザクション管理機能

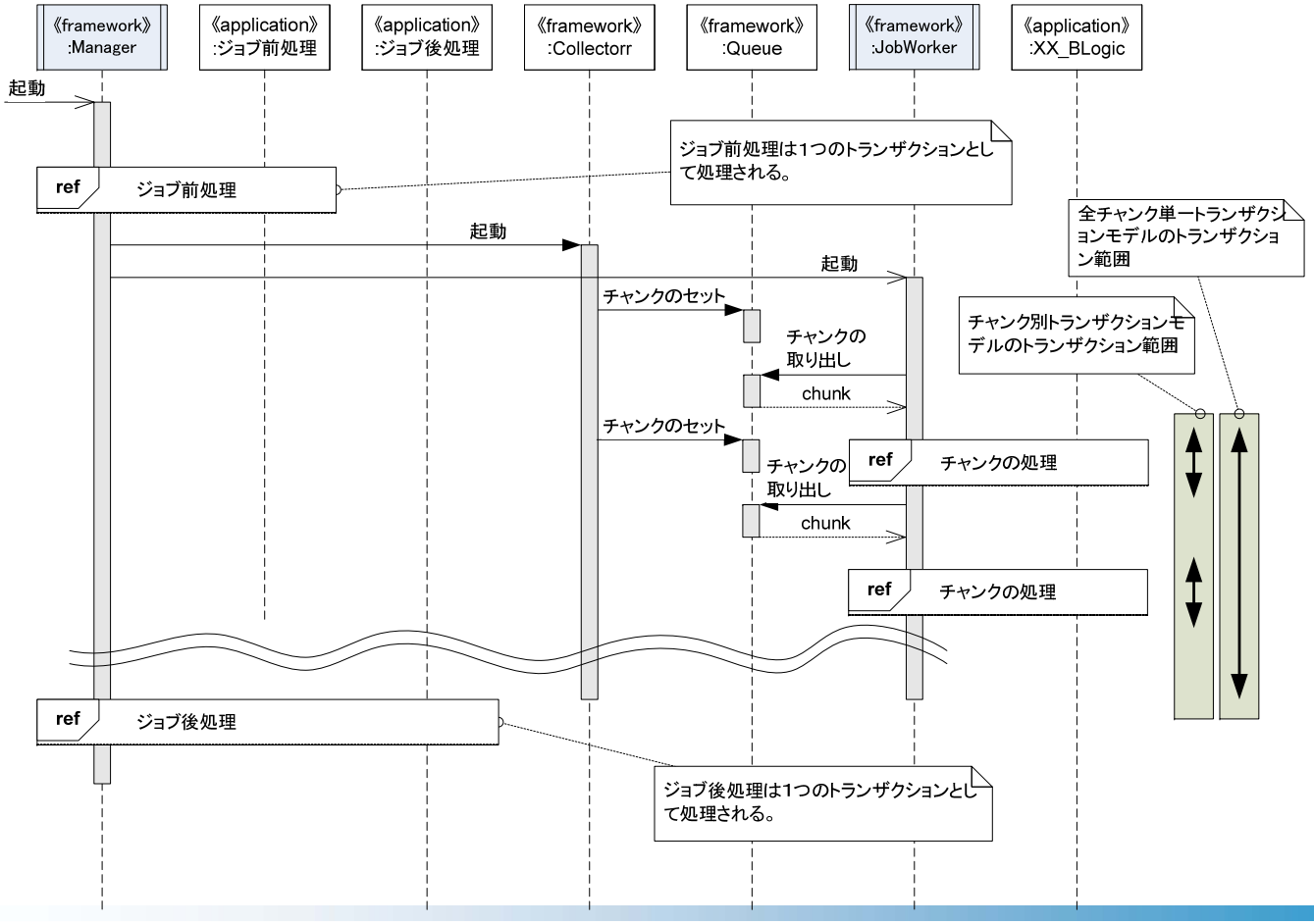
機能概要

- トランザクション機能を提供する。
  - 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。
- 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。（⇒チャンクサイズの指定）
- 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し、設定ファイルで指定する。

	トランザクションモデル	説明
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。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。

BA-01 トランザクション管理機能

- 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



BA-01 トランザクション管理機能

- 開発上の留意点
  - 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。
  - 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。
  - 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。

BA-01 トランザクション管理機能

- コーディングポイント

「ジョブBean定義ファイル」の設定例

```
<import resource="../ChunkTransactionBean.xml"/>

<!-- チャンクサイズの指定 -->
<bean id="chunkSize" class="java.lang.Integer">
    <constructor-arg value="50" />
</bean>
```

ジョブの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定義雛形ファイルをimportする

ジョブに適したチャンクサイズを指定する

	Bean定義雛形ファイル名	説明
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	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
6	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
7	PartitionNoTransactionBean.xml	非トランザクションモデル（分割ジョブ）用の雛形
8	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

BB-01 データベースアクセス機能

機能概要

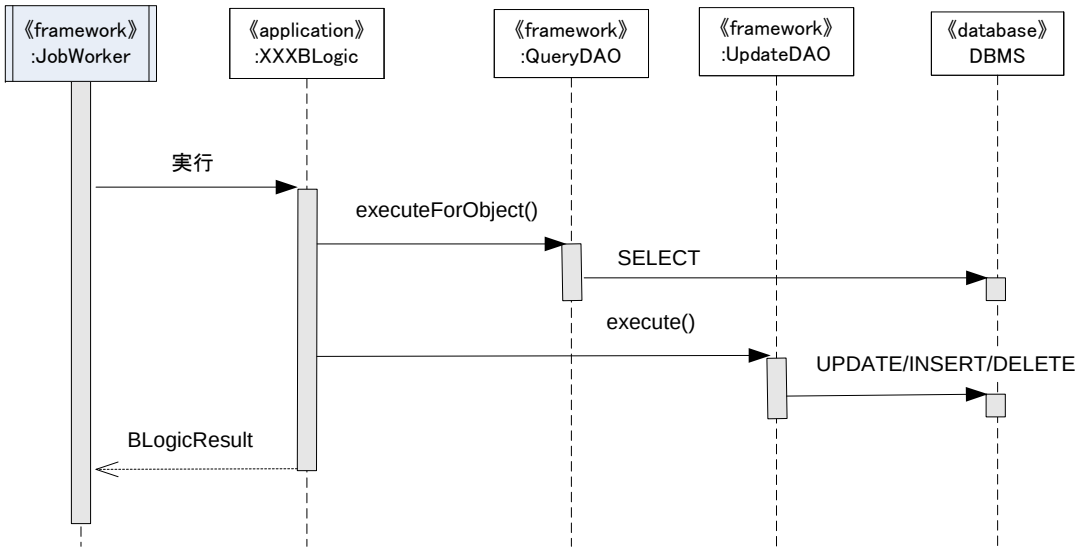
- データベースアクセスは、フレームワークで提供されるDAOを用いて行う。DAOインタフェースのデフォルト実装としてSpring ＋ iBATIS連携機能を利用したDAO実装を提供する。**（TERASOLUNA-OLTP版と共通）**
  - 開発者はiBATISの仕様に従って、設定ファイルにSQL文を定義する。

	DAOインタフェース	説明
1	QueryDAO	参照系のデータベースアクセスを行うDAOインタフェース
2	UpdateDAO	更新系のデータベースアクセスを行うDAOインタフェース
3	StoredProcedureDAO	ストアドプロシージャを実行するDAOインタフェース

- 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。**（TERASOLUNA-Batch版特有）**
  - 開発者は、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に登録してリターンする実装を、ビジネスロジック内で行なう。

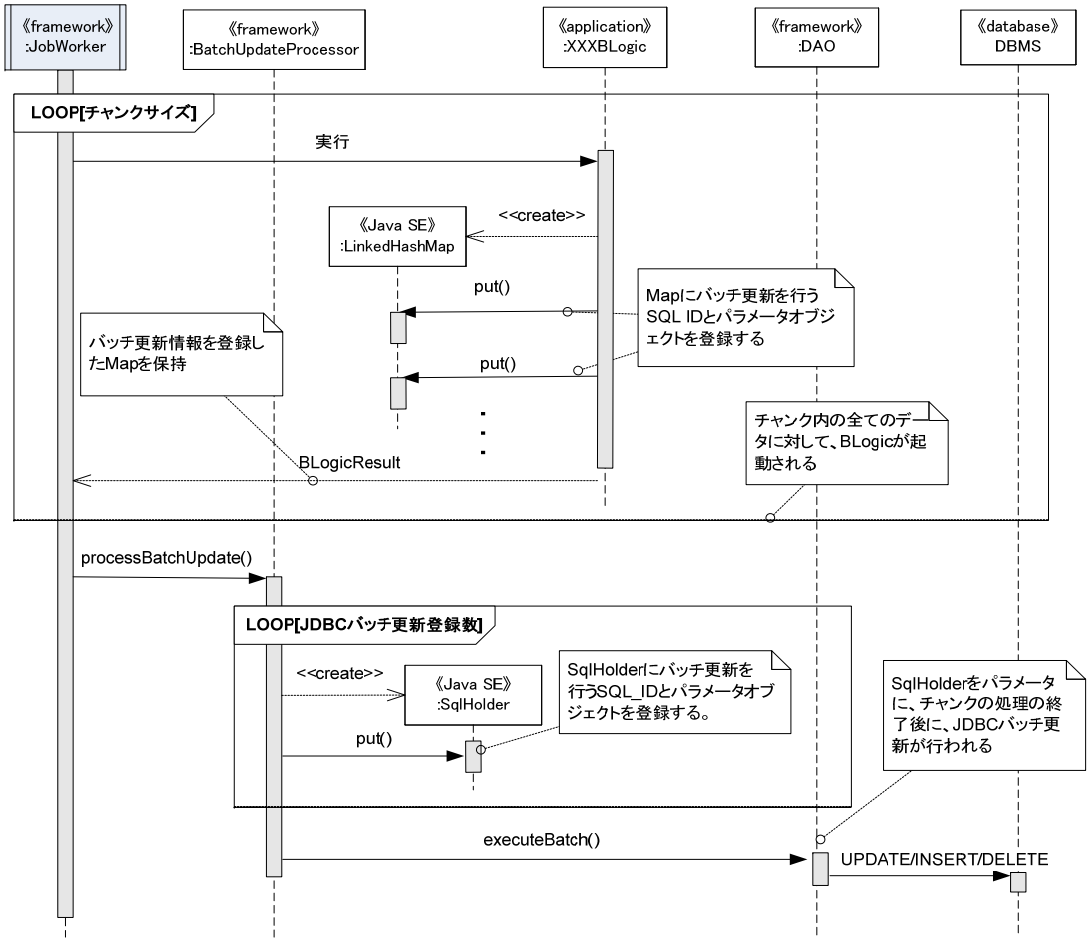
BB-01 データベースアクセス機能

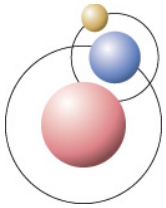
- 概念図
  - ビジネスロジックでは、フレームワークが提供するDAOによってデータベースアクセスを行うことができる



BB-01 データベースアクセス機能

- 複数のビジネスロジックにまたがったJDBCバッチ更新

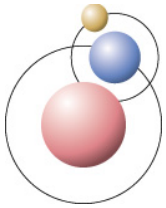




BB-01 データベースアクセス機能

- 開発上の留意点
  - 「ビジネスロジックをまたがったJDBCバッチ更新」について
    - 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。後続処理が、当該SQLの更新結果に依存するような場合には利用することができない。
    - 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
  - データベースアクセス時の例外
    - 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の例を示す。

	Spring例外クラス	説明
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。



BB-01 データベースアクセス機能

- コーディングポイント

「iBatis定義ファイル」の設定例

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

「ジョブBean定義ファイル」の設定例

```
<bean id="blogic  class="jp.terasoluna....XXXBLogic">
    <property name="updateDAO" ref="updateDAO" />
</bean>
...
```

「ビジネスロジック」の実装例

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


TERASOLUNA

BB-01 データベースアクセス機能

- コーディングポイント(続き)

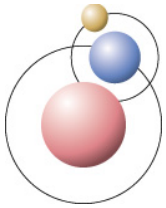
「ビジネスロジック」の実装例

JDBCバッチ更新の登録例

```
LinkedHashMap batchUpdateMap = new LinkedHashMap();
…
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

BLogicResut生成時にバッチ更新情報を格納したマップを渡す

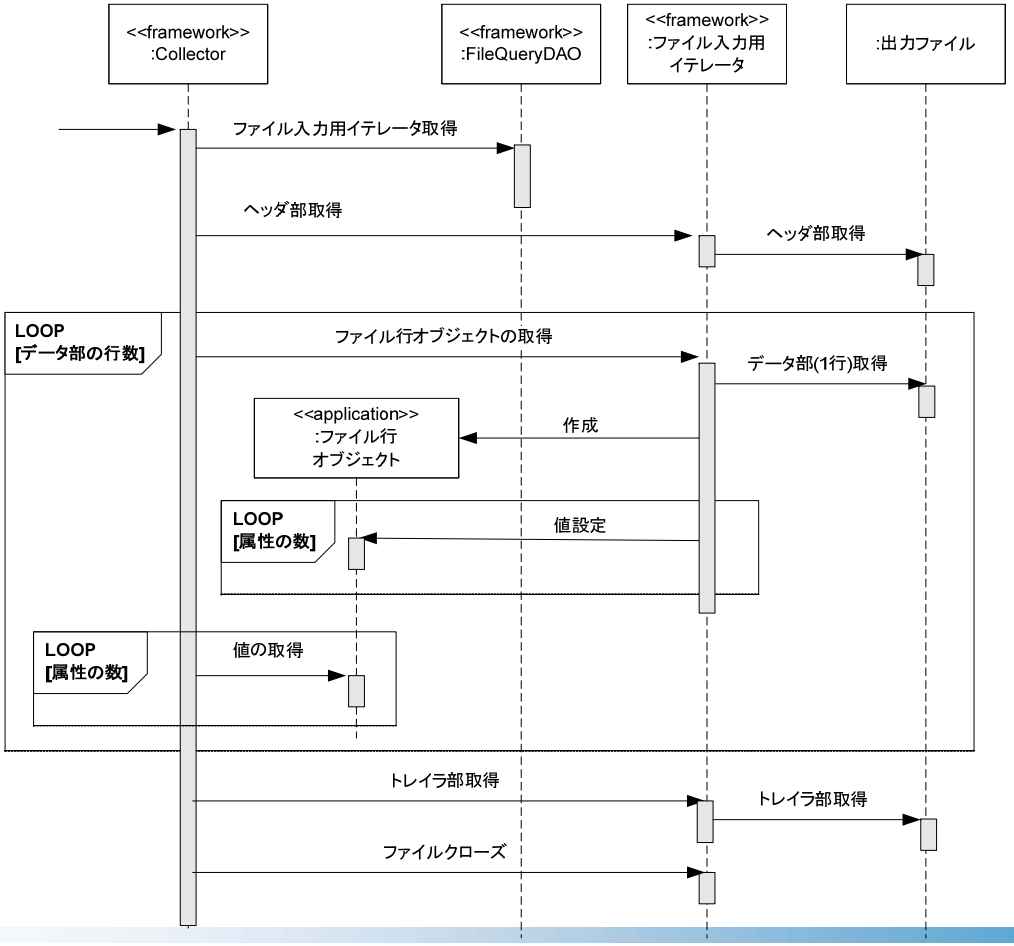


BC-01 ファイルアクセス機能

- 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
    - フレームワークではファイル入出用DAOのインタフェースを規定し、ファイル形式に対応したデフォルト実装を提供する。
    - 開発者は、ファイル名等、ファイルに関する情報をファイル入出力用DAOの属性に設定する。ファイル入出力用DAOの属性は「ジョブBean定義ファイル」で設定することができる。
  - 入力データに対するフォーマット処理を行う
    - 項目に対するパディング(Padding)、トリム（Trim）、文字変換（TextTransForm）等が指定できる
    - 開発者は「ファイル行オブジェクト」のアノテーションにより、フォーマットを定義する
  - 入力データに対する入力チェックを行う
    - 開発者は、「バリデーション定義ファイル(validation.xml)」の設定を行なう。また、ファイル入力用DAOの属性に「validator」プロパティを設定する

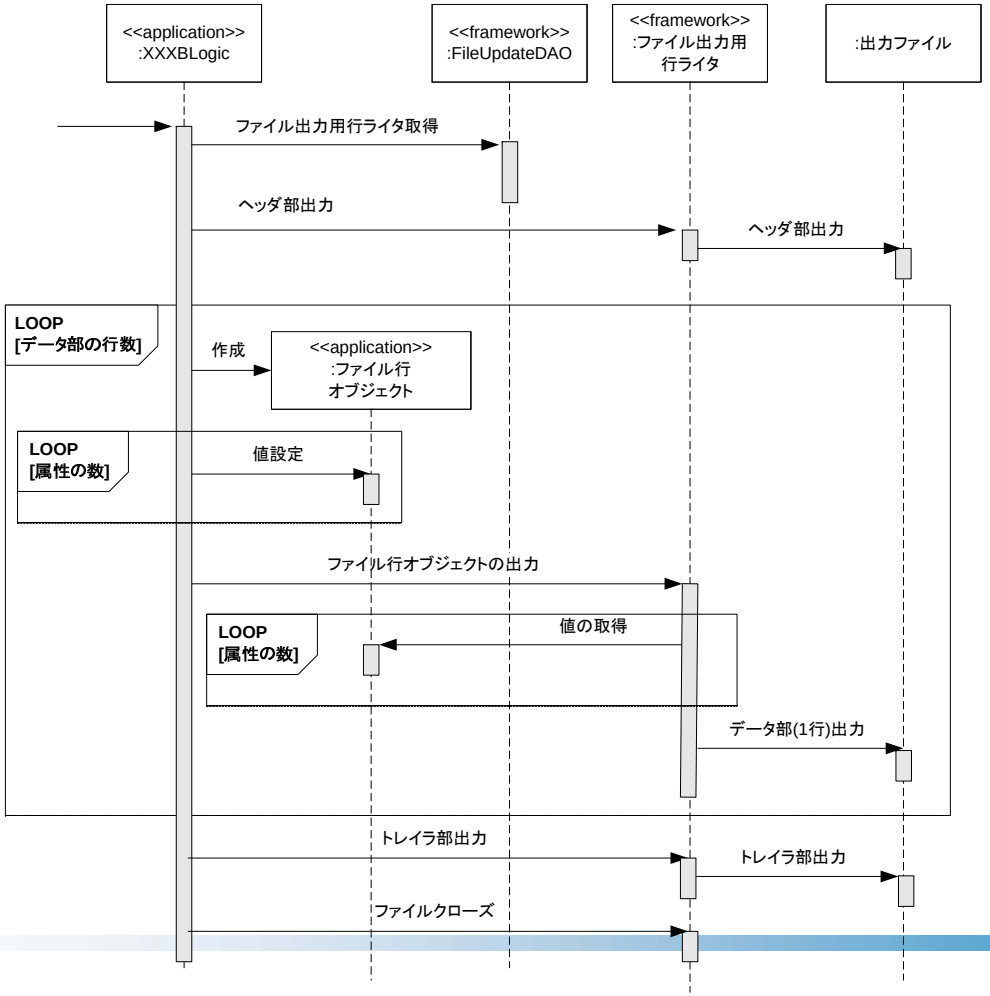
BC-01 ファイルアクセス機能

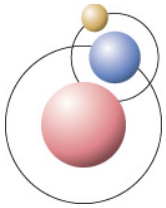
- 概念図
  - ファイル入力（Collectorから利用する場合）



BC-01 ファイルアクセス機能

- 概念図（続き）
  - ファイル出力（ビジネスロジックから利用する場合）





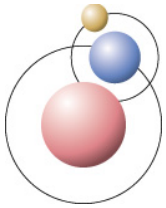
BC-01 ファイルアクセス機能

- 開発上の留意点
  - フレームワークで規定するファイル入力用DAOインタフェースと、ファイル形式に対応したデフォルト実装

	インタフェース名	説明
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

	クラス名	説明
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	ファイル行オブジェクトを利用せずにファイル入力を行なう場合に利用する

	クラス名	説明
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	ファイル行オブジェクトを利用せずにファイル入力を行なう場合に利用する



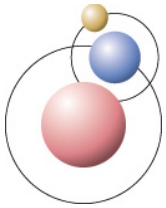
BC-01 ファイルアクセス機能

- 開発上の留意点（続き）
  - フレームワークで規定するファイル出力用DAOインタフェースと、ファイル形式に対応したデフォルト実装

	インタフェース名	説明
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

	クラス名	説明
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	ファイル行オブジェクトを利用せずにファイル出力を行なう場合に利用する

	クラス名	説明
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	ファイル行オブジェクトを利用せずにファイル出力を行なう場合に利用する



BC-01 ファイルアクセス機能

- 開発上の留意点（続き）
  - ファイル項目の定義情報

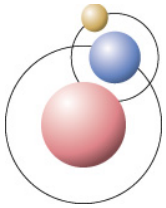
データ部の1行分のデータを格納するオブジェクトであり、ファイルのデータ項目に対応した属性を定義する。（使用できる型は、String、int、BigDecimal、Dateの4種類）

データ構造の定義は属性に対するアノテーションで記述する。

InputFileColumn,OutputFileColumn

	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）	なし		○		○		○
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・小文字変換・無変換を設定する。 StringConverterインタフェースの実装クラスを選択する StringConverterToUpperCase.class(大文字に変変換)/ StringConverterToLowerCase.class(小文字に変換)/ NullStringConverter.class(変換しない)	NullStringConverter.class（変換しない）	○	○	○	○	○	○

※ ◎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。



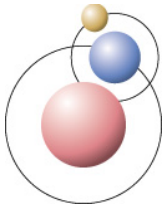
BC-01 ファイルアクセス機能

- 開発上の留意点（続き）
  - ファイル全体に関わる定義情報

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。

	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（上書きしない）		○		○		○		○

※ ○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 。



BC-01 ファイルアクセス機能

- コーディングポイント

「入力ファイル」の例

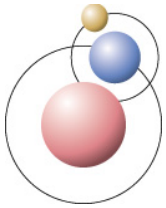
```
“2006/07/01”,”shop01”,”1,000,000”
```

「ファイル行オブジェクト」の実装例

```
@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;
      ......
}
```

「ファイル行オブジェクト」の属性値

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



BC-01 ファイルアクセス機能

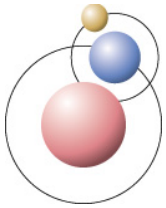
- コーディングポイント（続き）

「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



BC-01 ファイルアクセス機能

- コーディングポイント（続き）

「ビジネスロジック」の実装例

データ入力の例（ビジネスロジックでファイル入力を行なう場合）

```
...
  // 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",
                                        FileColumnSample.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

TERASOLUNA

BC-02 ファイル操作機能

- 機能概要
  - ファイル操作機能を提供する
    - ファイル名変更・ファイル移動／ファイルコピー／ファイル削除／ファイル結合
  - ファイル操作を行なうためのFileControlインタフェース および実装を提供する

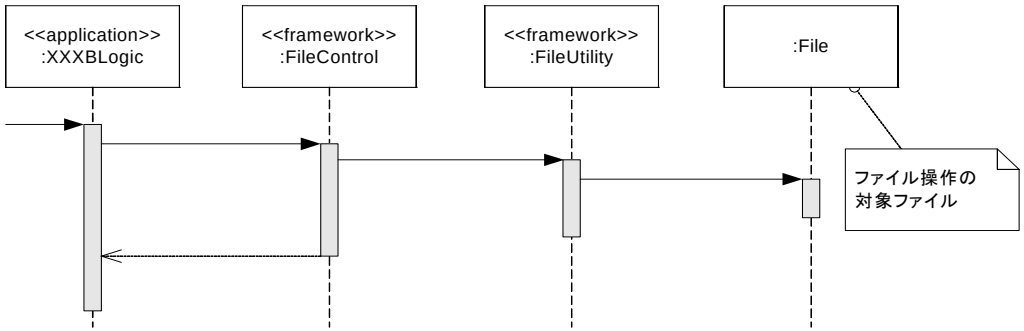
	インタフェース名	説明
1	jp.terasoluna.fw.file.util.FileControl	ファイル操作用のインタフェース

	代表的なメソッド	説明
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

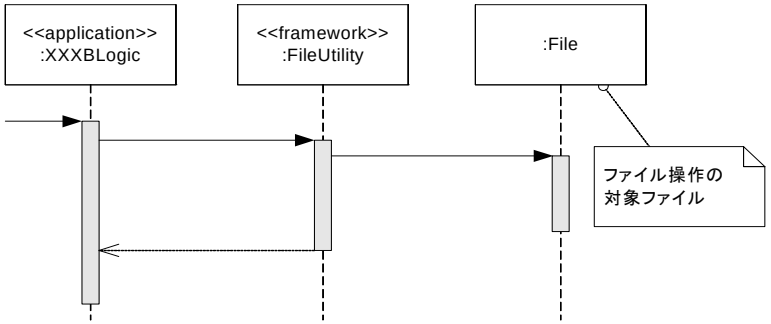
BC-02 ファイル操作機能

概念図

FileControl使用



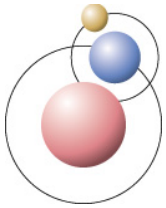
FileUtility使用



- フレームワークではファイル操作を行う際の基準となるパス（基準パス）を持つ
  - 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。

- ファイル操作ユーティリティクラス（FileUtility)を直接使用することも可能

	インタフェース名	説明
1	jp.terasoluna.fw.file.util.FileUtility	ファイル操作ユーティリティクラス



BC-02 ファイル操作機能

- コーディングポイント

「ジョブBean定義ファイル」の設定例

```
<bean id="sampleBLogic“   class="jp.terasoluna.batch.sample.SampleBLogic">
……
   <property name="fileControl">
       <ref bean="fileControl"/>
    </property>
```

ビジネスロジックのpropertyとして「fileControl」を設定する

「ビジネスロジック」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基準パスからの相対パスを記述する

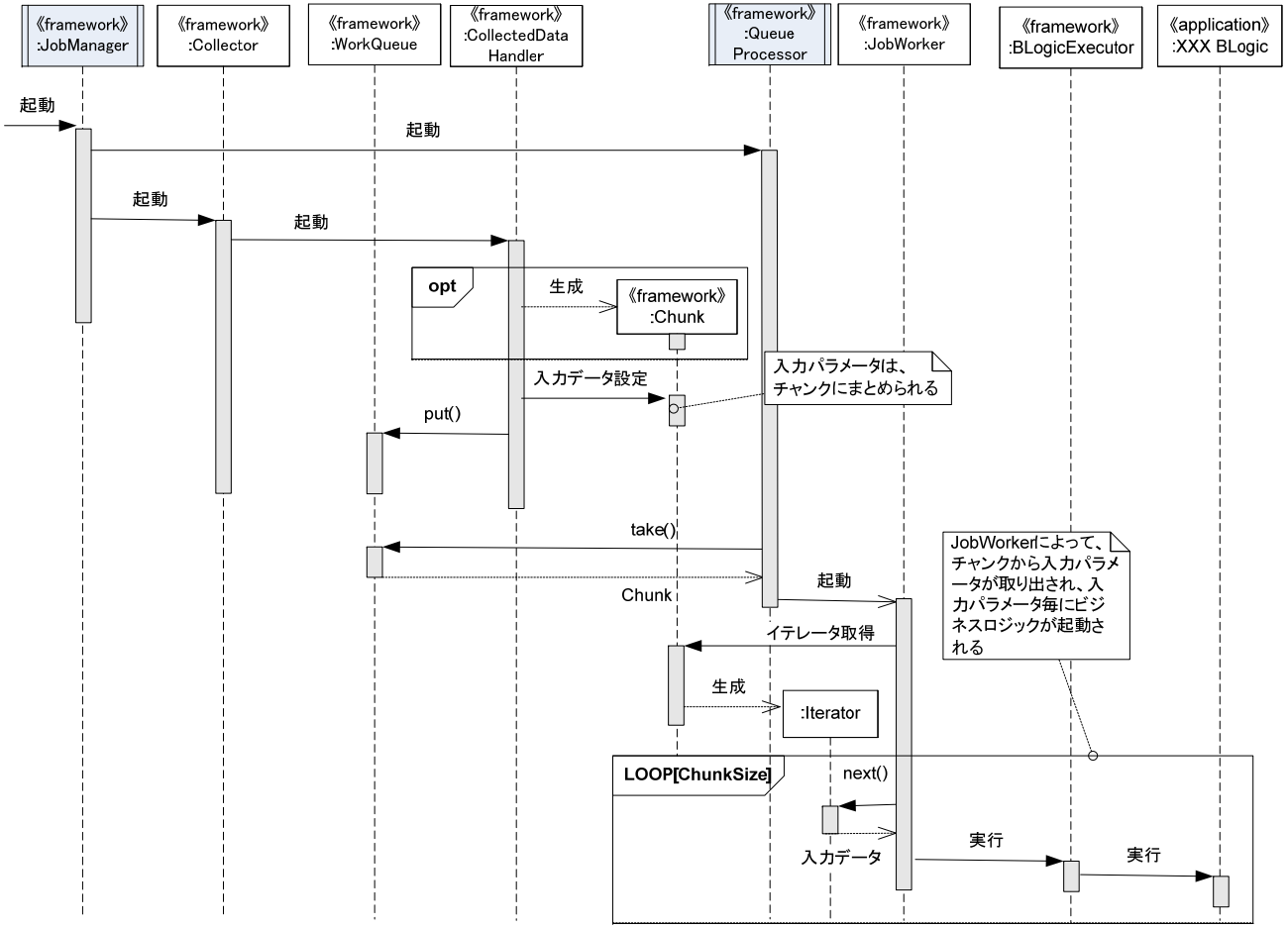
BD-01 ビジネスロジック実行機能

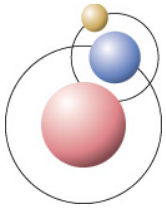
- 機能概要
  - ビジネスロジックの実行方式を提供する
    - 開発者は、TERASOLUNA-Batchが提供するBLogicインターフェースを実装してビジネスロジックを作成する。
  - ビジネスロジックの入力パラメータには、Collectorで作成したPOJOが渡される。
    - 開発者は、入力パラメータとして受け取るPOJOクラスを型パラメータとして定義する。
  - ビジネスロジックの実行結果としてBLogicResultを返却する。
    - 開発者は、BLogicResultのリターンコードにより、ジョブの継続/中止等をフレームワークに指示することができる。

	リターンコード	説明
1	NORMAL_CONTINUE	処理を継続する。通常はこのリターンコードを返却する。
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。
3	ERROR_CONTINUE	エラーデータとしてログ出力を行い、処理を継続する。
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。

BD-01 ビジネスロジック実行機能

- 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

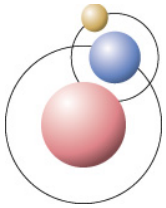




BD-01 ビジネスロジック実行機能

- 開発上の留意点
  - 入力パラメータ
    - 入力パラメータは、Collectorによって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。
    - ビジネスロジックへの入力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。例えばヘッダー明細の単位で論理トランザクションとして処理しなければならない場合などがある。
  - ビジネスロジック間で引き継ぐ情報、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。
    - 開発者は、フレームワークで規定する基底クラスを拡張してジョブコンテキストを作成することができる。

	基底クラス	説明
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クラス



BD-01 ビジネスロジック実行機能

- 開発上の留意点（続き）
  - ジョブに対しては、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。
    - 開発者は、フレームワークで規定するインタフェースを実装して前処理／後処理を作成する。
    - ジョブ前処理／ジョブ後処理は、すべてのトランザクションモデルで作成できる。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 である場合のみ作成できる。
    - 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。

	インタフェース	説明
1	jp.terasoluna.fw.batch.openapi.SupportLogic	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が実装する必要のあるインタフェース

BD-01 ビジネスロジック実行機能

- コーディングポイント

「ジョブBean定義ファイル」の設定例

```
...
<bean id="blogic"  class="jp.terasoluna....XXXBLogic">
    <property name="queryDAO" ref="queryDAO" />
    <property name="updateDAO" ref="updateDAO" />
</bean>
...
<bean id="jobContext"  class="jp.terasoluna....XXXJobContext"/>
...
<util:list id="jobPreLogicList">
    <bean class="jp.terasoluna....XXXPreLogic"/>
        ...
</util:list>
...
<util:list id="jobPostLogicList">
    <bean class="jp.terasoluna....XXXPostLogic"/>
        ...
</util:list>
```

ビジネスロジックを設定する

ビジネスロジックで利用するDAOを設定する

必要に応じてジョブコンテキストを設定する。ビジネスロジック、ジョブ前処理、ジョブ後処理で共有される。

ジョブ前処理を設定する。
必要がなければ、設定しなくてよい。

ジョブ後処理を設定する。
必要がなければ、設定しなくてよい。

BD-01 ビジネスロジック実行機能

- コーディングポイント（続き）

「ビジネスロジック」の実装例

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

    public BLogicResult execute(XXXInputData inputData,
                                XXXJobContext jobContext) {
        ...
        Date unyuHdk = jobContext.getUnyuHdk();
        ...
        if (isErrorData) {
            ...
            monitoringLog.error(...);

            return new BLogicResult(ReturnCode.ERROR_CONTINUE);
        }
        ...
        if (isError) {
            ...
            monitoringLog.error(...);

            return new BLogicResult(ReturnCode.ERROR_END, 2);
        }
        ...
        return new BLogicResult(ReturnCode.NORMAL_CONTINUE);
    }
}
```

BLogicインタフェースを実装し、型パラメータを指定する

前処理などで設定したデータを、ジョブコンテキストから取得できる。

エラーデータとして判定し、後続のデータの処理を続ける場合には、ログへ出力しERROR_CONTINUEをリターンする。

ジョブを終了する場合には、ジョブ終了コードを任意で設定し、ERROR_ENDをリターンする。

次の入力データの処理を行う場合には、NORMAL_CONTINUEをリターンする

BD-01 ビジネスロジック実行機能

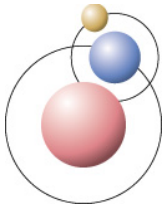
- コーディングポイント（続き）

「ジョブコンテキスト」の実装例

```
public class XXXJobContext extends JobContext {

        private Date UnyuHdk = null;

        public void set UnyuHdk(Date date) {
              this. UnyuHdk = date;
        }

        public Date get UnyuHdk() {
              return UnyuHdk;
        }

}
```

JobContextを継承し拡張することができる。
拡張したJobContextを使う場合はジョブBean定義ファイルに定義する必要がある。
ジョブBean定義ファイルにJobContextの定義がない場合はデフォルトのJobContextが使用される。

業務APで必要な属性および、getter/setterを記述する



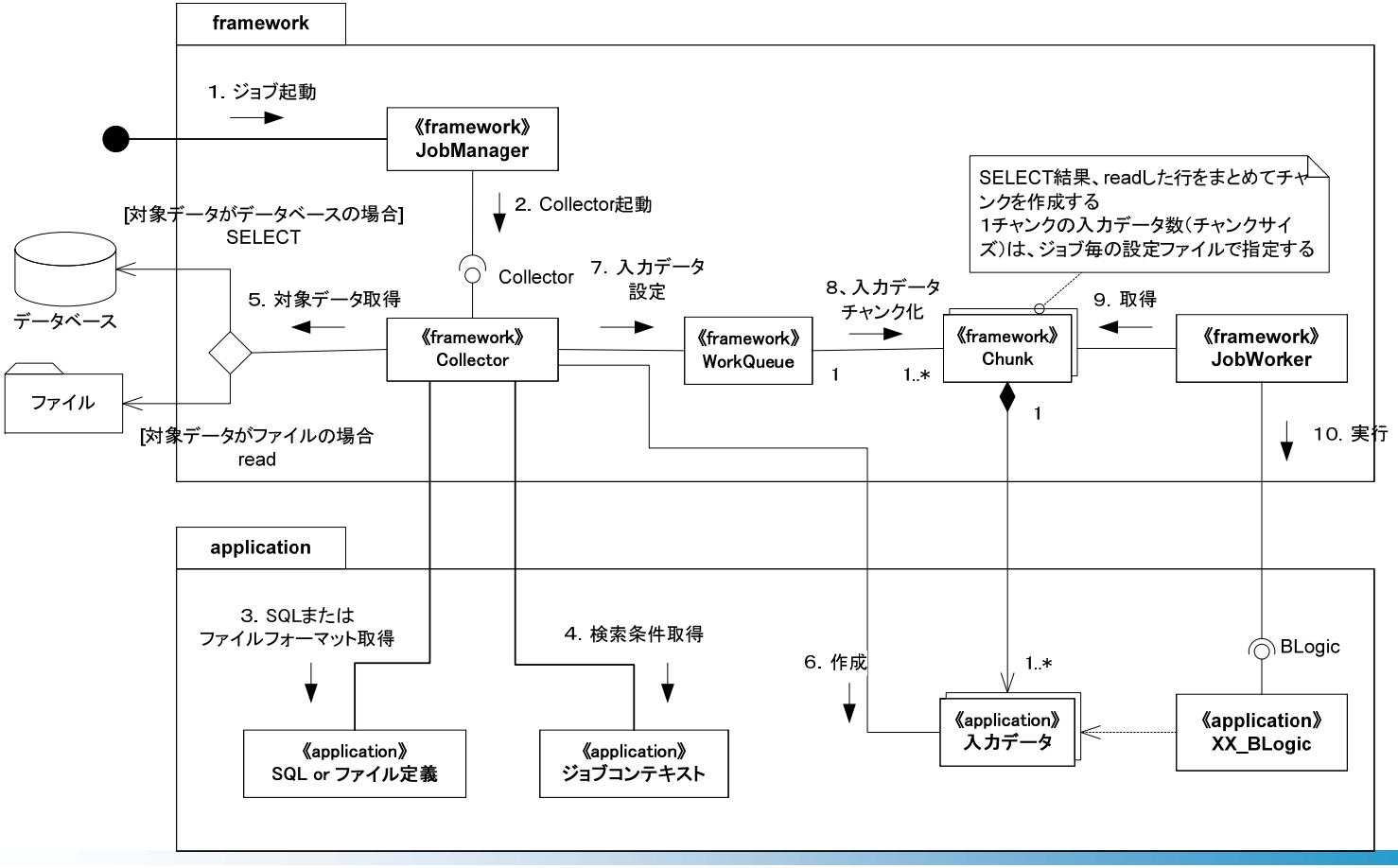
BD-02 対象データ取得機能

- 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。
  - 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
    - 開発者は、TERASOLUNA-Batchが提供するCollectorインタフェースとそのpropertyを、ジョブBean定義ファイルに設定する

	Collectorデフォルト実装	説明
1	jp.terasoluna.fw.batch.ibatissupport.IBatisDbCollectorImpl	iBATISデータベースCollector 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	jp.terasoluna.fw.batch.standard.StandardFileCollector	ファイルCollector 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	jp.terasoluna.fw.batch.standard.ListPropertyCollector	リストプロパティCollector 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	jp.terasoluna.fw.batch.standard.StringArrayPropertyCollector	文字列配列プロパティCollector 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

BD-02 対象データ取得機能

- 概念図
  - 対象データ取得の静的構造



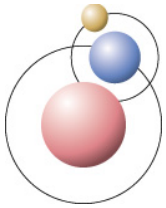
BD-02 対象データ取得機能

- 開発上の留意点
  - Collectorは対象データの取得が終了すると、CollectorResultを返却する。
    CollectorResultは、以下のリターンコードを持つ

	リターンコード	説明
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。

  - 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。

	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



BD-02 対象データ取得機能

- 開発上の留意点（続き）
  - データ取得の条件の指定
    - Collector実装にはジョブコンテキストが渡される。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。
  - 対象データが0件であったときの扱い
    - 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。
  - 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
  - 対象データ取得でエラーが発生した場合の扱い
    - 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

BD-02 対象データ取得機能

- コーディングポイント

「ジョブBean定義ファイル」の設定例

iBATISデータベースCollectorの設定

```
<!--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

「iBatis定義ファイル」の設定例

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

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。

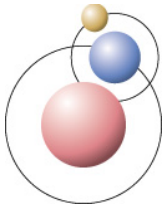
BD-02 対象データ取得機能

- コーディングポイント（続き）

「ビジネスロジック」の実装例

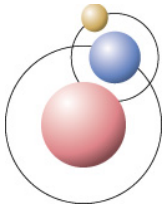
```
public class XXXBLogic implements BLogic<XXXInputData, SampleJobContext> {
   private QueryDAO queryDAO;
   private UpdateDAO updateDAO;

   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



BD-03 コントロールブレイク機能

- 機能概要
  - 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。
  - 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。
    - 開発者はジョブBean定義ファイルにブレイクキーとそれに対応するコントロールブレイクハンドラの設定を行い、コントロールブレイクハンドラの実装を行う



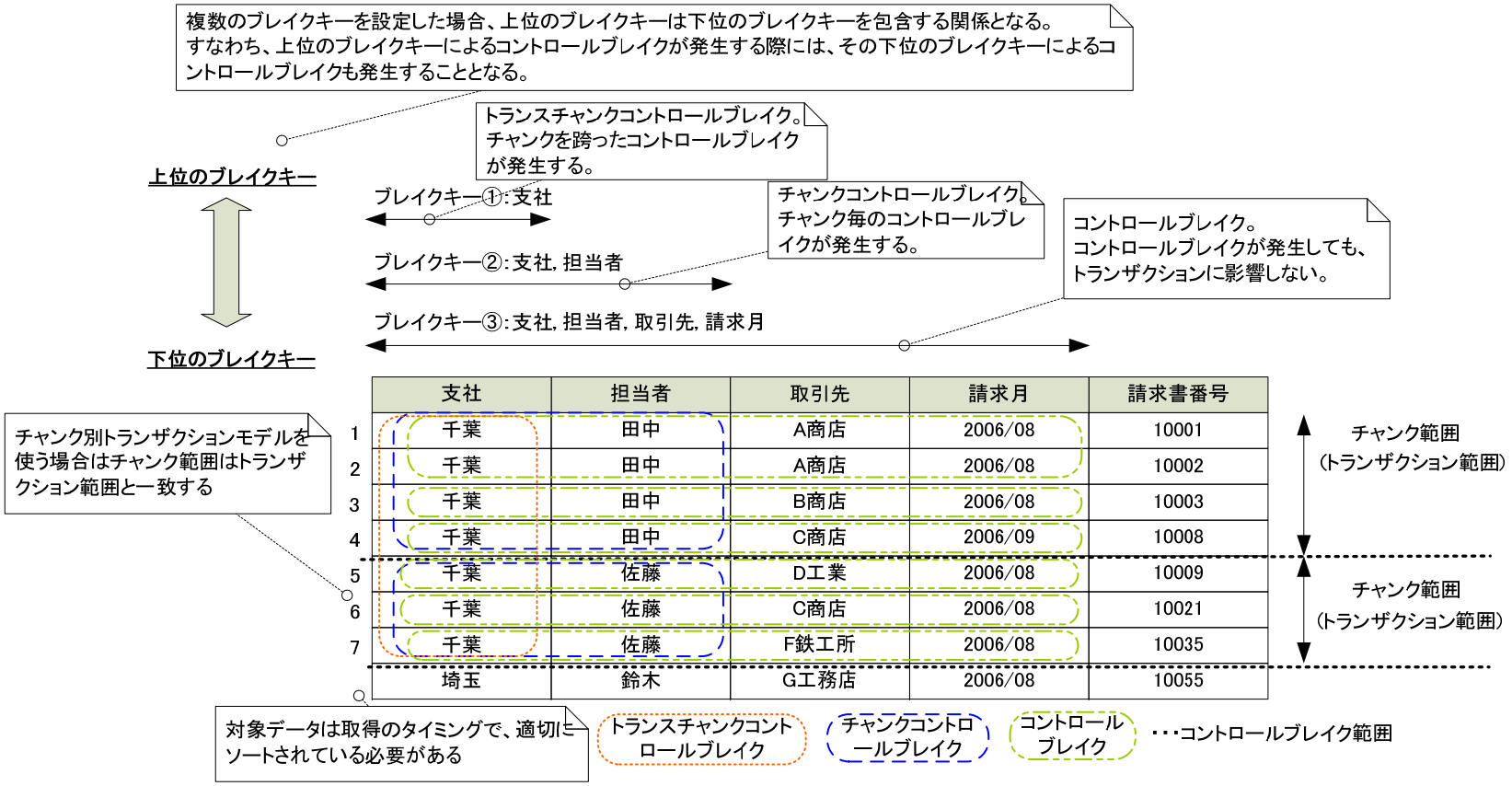
BD-03 コントロールブレイク機能

- 機能概要（続き）
  - 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

	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。

BD-03 コントロールブレイク機能

- 概念図
  - 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

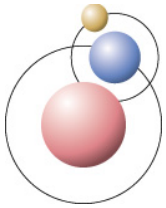


	支社	担当者	取引先	請求月	請求書番号
1	千葉	田中	A商店	2006/08	10001
2	千葉	田中	A商店	2006/08	10002
3	千葉	田中	B商店	2006/08	10003
4	千葉	田中	C商店	2006/09	10008
5	千葉	佐藤	D工業	2006/08	10009
6	千葉	佐藤	C商店	2006/08	10021
7	千葉	佐藤	F鉄工所	2006/08	10035
	埼玉	鈴木	G工務店	2006/08	10055

BD-03 コントロールブレイク機能

- 概念図（続き）
  - 概念図の例における処理の流れ

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
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



BD-03 コントロールブレイク機能

- 開発上の留意点
  - 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。

	インタフェース	説明
1	jp.terasoluna.fw.batch.openapi.ControlBreakHandler	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。

	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。

BD-03 コントロールブレイク機能

- 開発上の留意点（続き）
  - 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。

	入力パラメータ	説明
1	キー値リスト	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。
2	ジョブコンテキスト	そのジョブのジョブコンテキスト。

  - 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。

BD-03 コントロールブレイク機能

- コーディングポイント

「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 class="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>
```

コントロールブレイク用の雛形を設定する。

コントロールブレイク用Collector設定 。

チャンク内で発生する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。

BD-03 コントロールブレイク機能

- コーディングポイント（続き）

「ジョブBean定義ファイル」の設定例（続き）

```
<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>

<util:list id="transChunkControlBreakDefItemList>
        <bean class="jp.terasoluna.fw.batch.controlbreak.ControlBreakDefItem">
            <property name="breakKey">
                <list>
                    <value>branchCd</value>
                </list>
            </property>
            <property name="contorlBreakHandler">
                <bean class="jp.terasoluna.....XXXControlBreakHandler">
                    <property name=”queryDAO” ref=”queryDAO”>
                </bean>
            </property>
        </bean>
</util:list>
・・・
```

チャンクコントロールブレイクの設定。

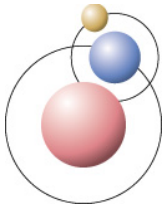
指定したキーによりチャンクが作成される。

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。

トランスチャンクコントロールブレイクの設定。

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。

必要に応じてコントロールブレイクハンドラのプロパティをDIで設定する。



BD-03 コントロールブレイク機能

- コーディングポイント（続き）

「コントロールブレイクハンドラ」の実装例

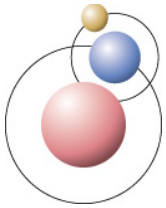
```
pubic class TantosyaSummaryHandler implements ControlBreakHandler<MyJobContext> {
  private QueryDAO queryDAO;
    ...

  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。

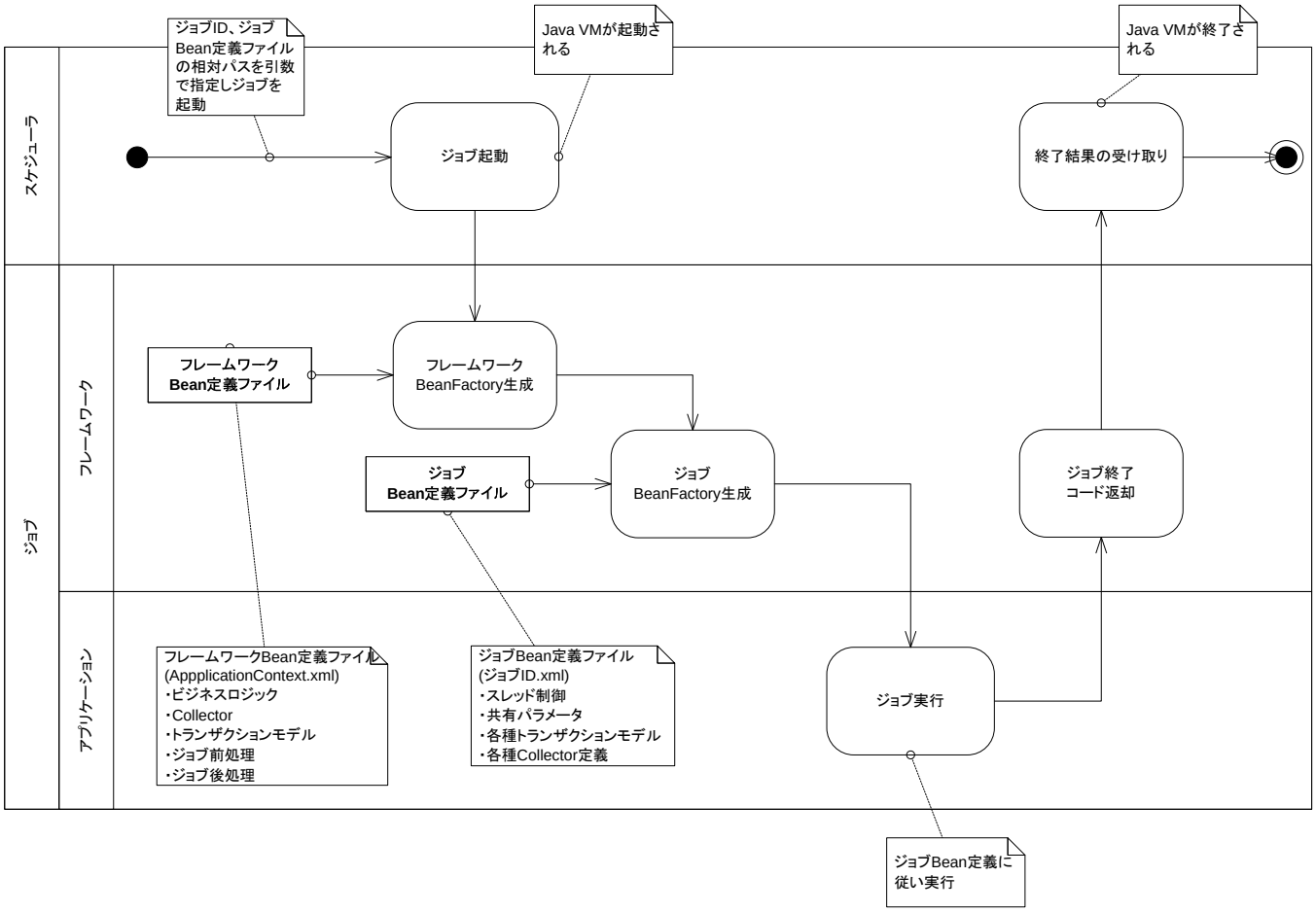


BE-01 同期型ジョブ起動機能

- 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
  - 起動するジョブの定義はジョブ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

BE-01 同期型ジョブ起動機能

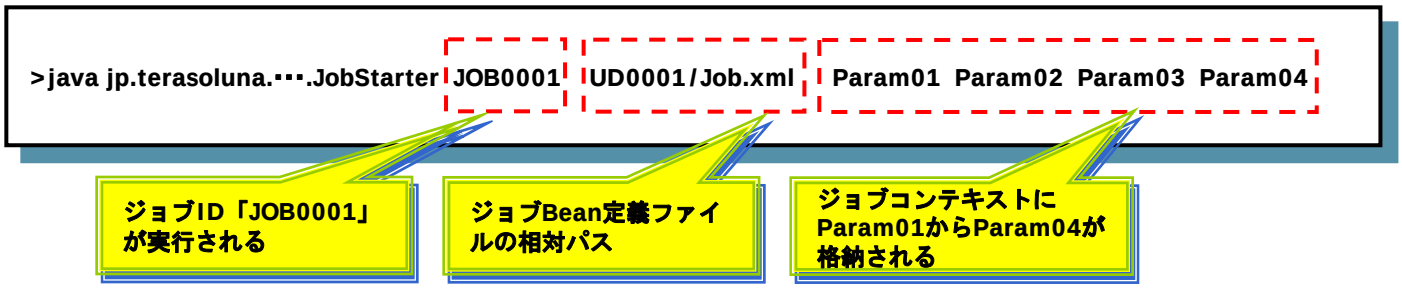
概念図



BE-01 同期型ジョブ起動機能

- フレームワークの起動例
  - ジョブ起動時の引数
    - 第1引数：起動するジョブID（必須）
    - 第2引数：ジョブBean定義ファイルの相対パス（必須）
    - 第3引数以降：パラメータ値（任意）

フレームワーク起動のコマンド入力例



BE-01 同期型ジョブ起動機能

- コーディングポイント(ジョブコンテキスト)

フレームワーク起動のコマンド入力例

```
>java jp.terasoluna.･･･.JobStarter  JOB0001  UD0001/Job.xml  会社名 開始日 終了日  testParam.properties
```

パラメータ用引数

パラメータ用引数(ファイル)

ジョブコンテキストの記述例

```
public class SampleJobParameter extends JobContext {
      private String company = null;
      private Date startDay = null;
      private Date endDay = null;
      private List<String> fileData = null;
      public void setParameter(String[] arg) {
          company = arg[0];
          startDay = DateFormat.getTimeInstance().parse(arg[1]);
          endDay = DateFormat.getTimeInstance().parse(arg[2]);
          Properties p = new Properties();

          FileInputStream fis = new FileInputStream(arg[3]);
          p.load(fis);
      ・・・・・
}
 ・・・・・
```

引数からのパラメータ情報格納

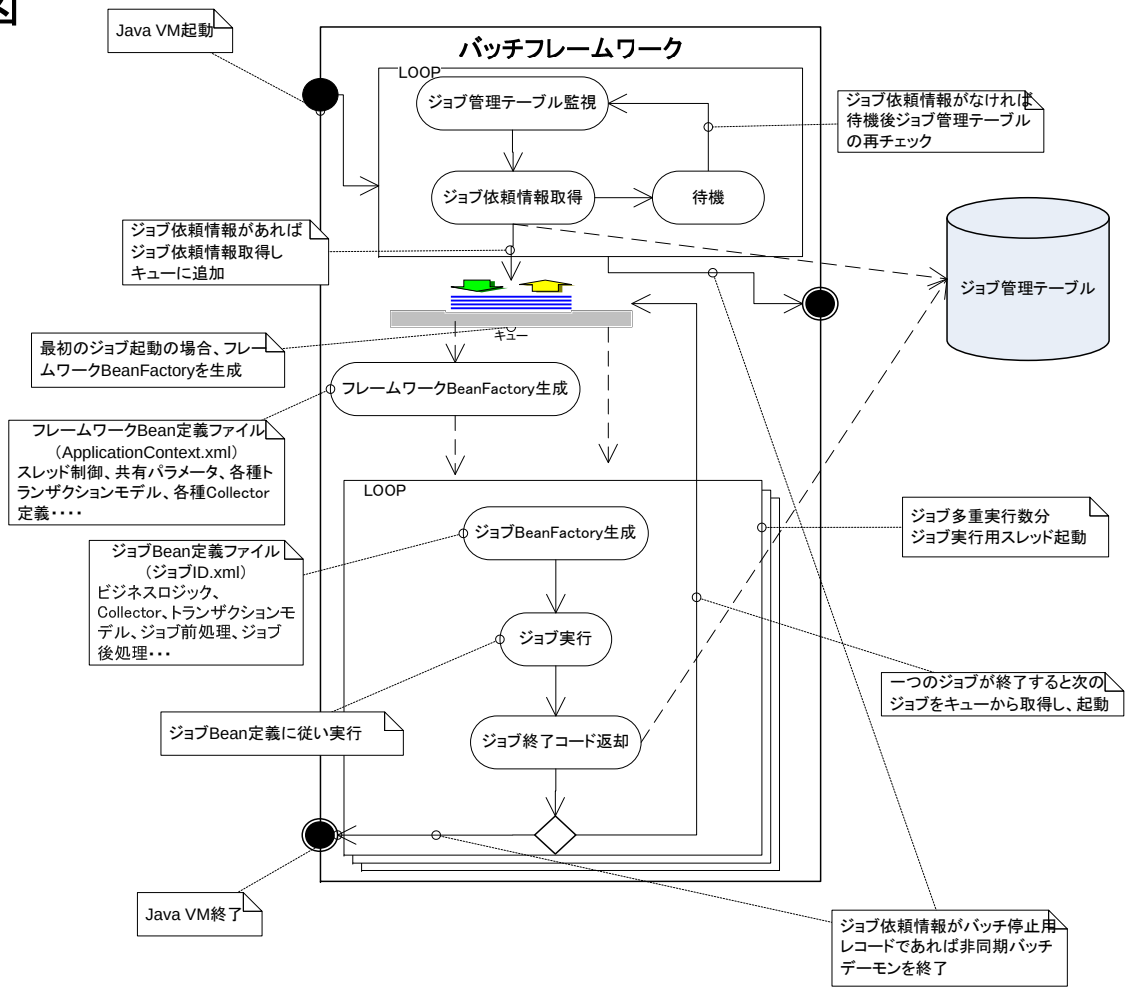
ファイルからのパラメータ情報格納

BE-02 非同期型ジョブ起動機能

- 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起動される
    - 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)で指定することができる
  - 非同期バッチジョブ起動機能では、下記のバッチ種別でのジョブ起動が可能である
    - スケジュールバッチ
    - 周期バッチ
    - 随時バッチ

BE-02 非同期型ジョブ起動機能

- 概念図



BE-02 非同期型ジョブ起動機能

- 非同期バッチデーモンの起動例

非同期バッチデーモン起動のコマンド入力例

```
> java jp.terasoluna.fw.batch.springsupport.init.AsyncBatchDaemon
```

「AsyncBatchDaemonBean.xml」の定義内容に従い起動

BE-02 非同期型ジョブ起動機能

- ジョブ管理テーブルとは
  - 実行するジョブの情報を管理するテーブルである
    - 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

	属性名	カラム名	Not Null	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。
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(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

BE-03 ジョブ実行管理機能（一部非推奨機能）

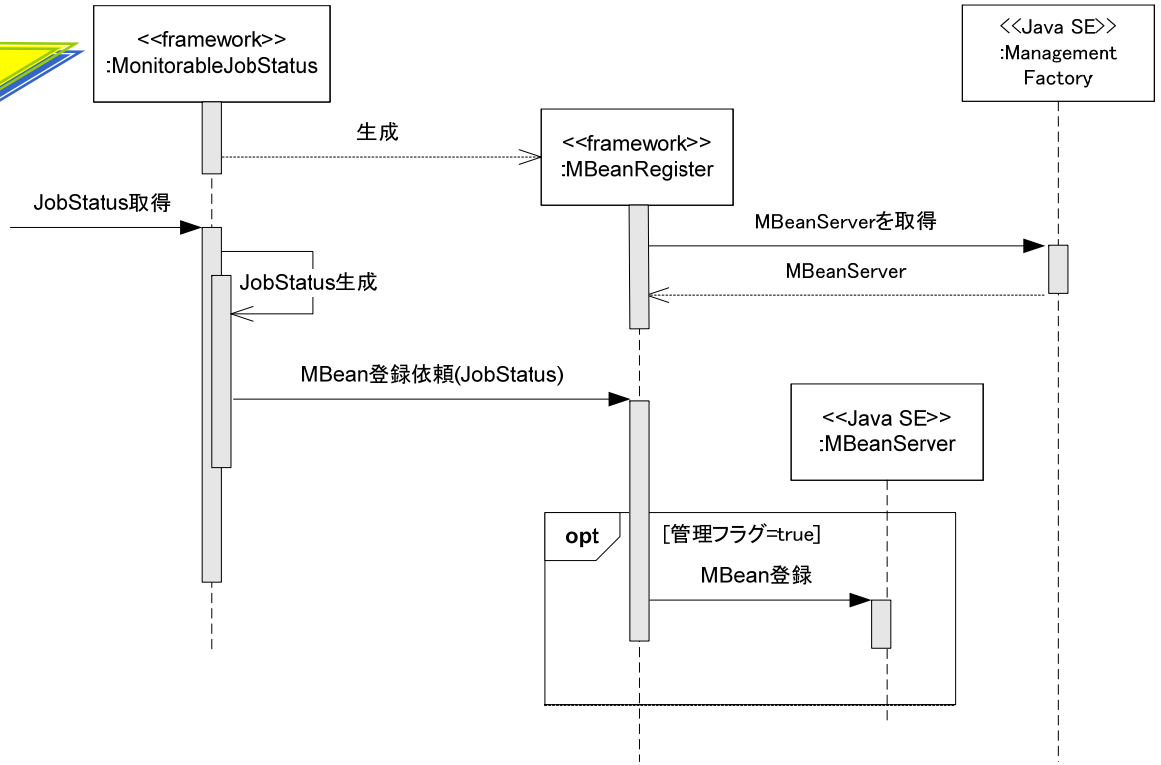
- 機能概要
  - JMXを使用したジョブ監視機能を提供する
    - MBean登録クラス(MBeanRegister)はバッチジョブ起動クラスから生成され、MBeanServerの取得を行う
    - MBeanはMBean登録クラスを介してMBeanServerに登録することにより、監視が可能となる
    - 監視ツールはjconsoleを使用する
      - ※jconsoleとは、Java SE5.0に付属するJMX準拠の監視ツールである
  - ジョブ終了制御機能を提供する
    - ジョブが実行中の場合のみ終了が可能
    - 終了には強制終了と中断終了がある
      - 強制終了
        - » 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
        - » 次のコミットポイントまで処理を継続し、コミット終了後にジョブを終了する
    - 監視ツール(jconsole)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

※JMXを利用したジョブ監視機能についてはTERASOLUNA Batch Framework for Java 2.0.2.0より非推奨機能となります。ジョブの監視についてジョブスケジューラの機能をご利用下さい。また終了・中断ファイルによるジョブ終了・中断機能は非推奨機能ではございませんのでそのままご利用ください。

BE-03 ジョブ実行管理機能（一部非推奨機能）

- 概念図
  - JMXで管理するリソースは、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り、JMXによる監視を行う場合にはJobStatusの替わりにこのクラスを使うようにSpringのフレームワークBean定義ファイルで指定する。



BE-03 ジョブ実行管理機能（一部非推奨機能）

- コーディングポイント
  - 下記の設定はソフトウェアアーキテクトが行う

「フレームワークBean定義ファイル」の設定例

ジョブ管理に関する設定

```
<bean id=“MBeanRegister” class="jp.terasoluna.batch.fw.mbean.MBeanRegister">
    <property name="jmxEnable">
        <value>true</value>
    </property>
     <property name=”manageableJobSize” value=“10” />
</bean>

<bean id="JobStatus" class="jp.terasoluna.fw.batch.monitor.MonitorableJobStatus">
     <property name="mbeanRegister" ref="MBeanRegister"/>
</bean>
```

ジョブ管理の有無

監視ツールで管理可能なジョブの上限値

監視用のJobStatus

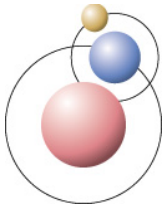
「フレームワークBean定義ファイル」の設定例

ジョブ終了制御に関する設定例

```
<bean id="endFileCheck" class="jp.terasoluna.fw.batch.init.EndFileCheck">
    <propertyname="endFileDir“ value=“batchapps/BE-03/EndFile” />
</bean>

<beanid="scheduledTask"  class="org.springframework.scheduling.timer.ScheduledTimerTask">
   <property name="period" value="8000"/>
   <property name="timerTask" ref="endFileCheck"/>
</bean>
```

終了ファイル検出ディレクトリ

チェック間隔（秒）

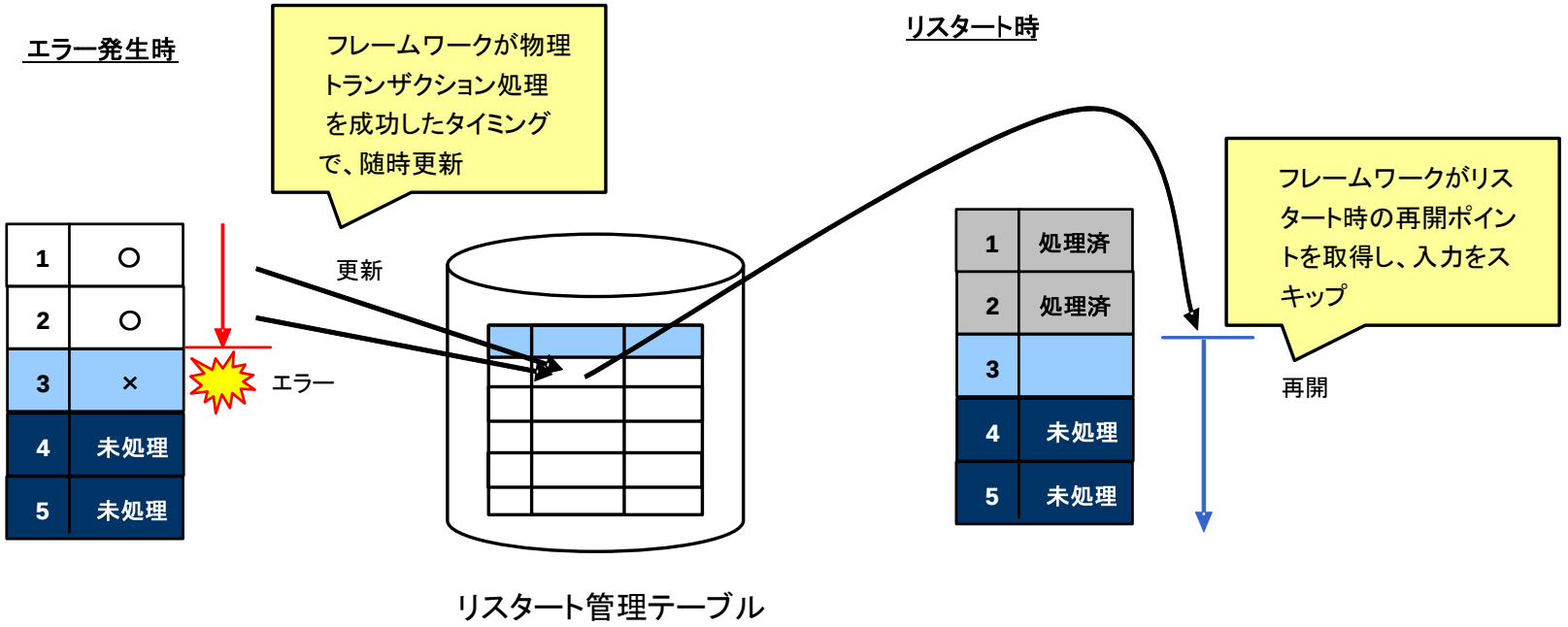


BE-04 リスタート機能

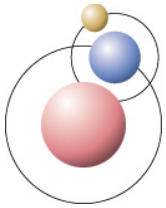
- 機能概要
  - ジョブのリスタート機能を提供する
    - ジョブ実行時にリスタート管理テーブルにリスタートポイントの記録を行ない、中断ジョブの再開時にはリスタートポイント以降のデータから処理を再開する
      - リスタートポイントとは、処理を完了（コミット）したレコード数
    - フレームワークは以下の処理を提供する
      - リスタートポイント、ジョブコンテキストの登録･更新
      - リスタート時のジョブコンテキストの復元処理
      - リスタート時のリスタートポイント以降の対象データ取得処理
      - リスタート情報のクリア処理
    - 開発者は、設定ファイルにおいてリスタート機能を有効とすることにより、リスタート可能ジョブとして定義することができる。
    - 開発者は、対象データ取得において、リスタート時でも同一のデータが取得できるように、SQL文を記述する必要がある
    - ソフトウェアアーキテクトは、リスタート機能を利用するための環境設定や規約作成を行なう
      - リスタート管理テーブルの作成
      - ジョブ起動パラメータの決定、SQL文コーディング規約

BE-04 リスタート機能

概念図



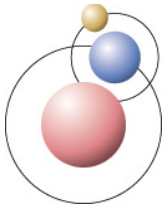
- ジョブが中断されるまでの情報をリスタート管理テーブルで管理する
- フレームワークがジョブを再開(リスタート)する際、リスタート管理テーブルからジョブコンテキストなどを復元する



BE-04 リスタート機能

- 開発上の留意点
  - リスタート機能使用時の注意事項
    - 業務アプリケーションにおいて、リスタート可能な設計（ステータスカラムや処理済フラグを利用する等）を行なっている場合には、本機能を利用する必要は無い。
    - リスタート機能を利用できるトランザクションモデルは以下のとおり
      - チャンク別トランザクションモデル
      - チャンク別トランザクションモデル（分割ジョブ）

      上記のモデル以外でも雛形Bean定義を作成することで他モデルへの適用も可能
    - 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を対象外とする。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
    - 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、本機能を利用することはできない。



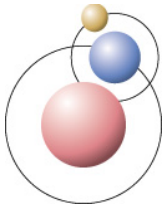
BE-04 リスタート機能

- 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

```
...
<!-- 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" />
  ...
```

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。



BE-05 処理結果ハンドリング機能

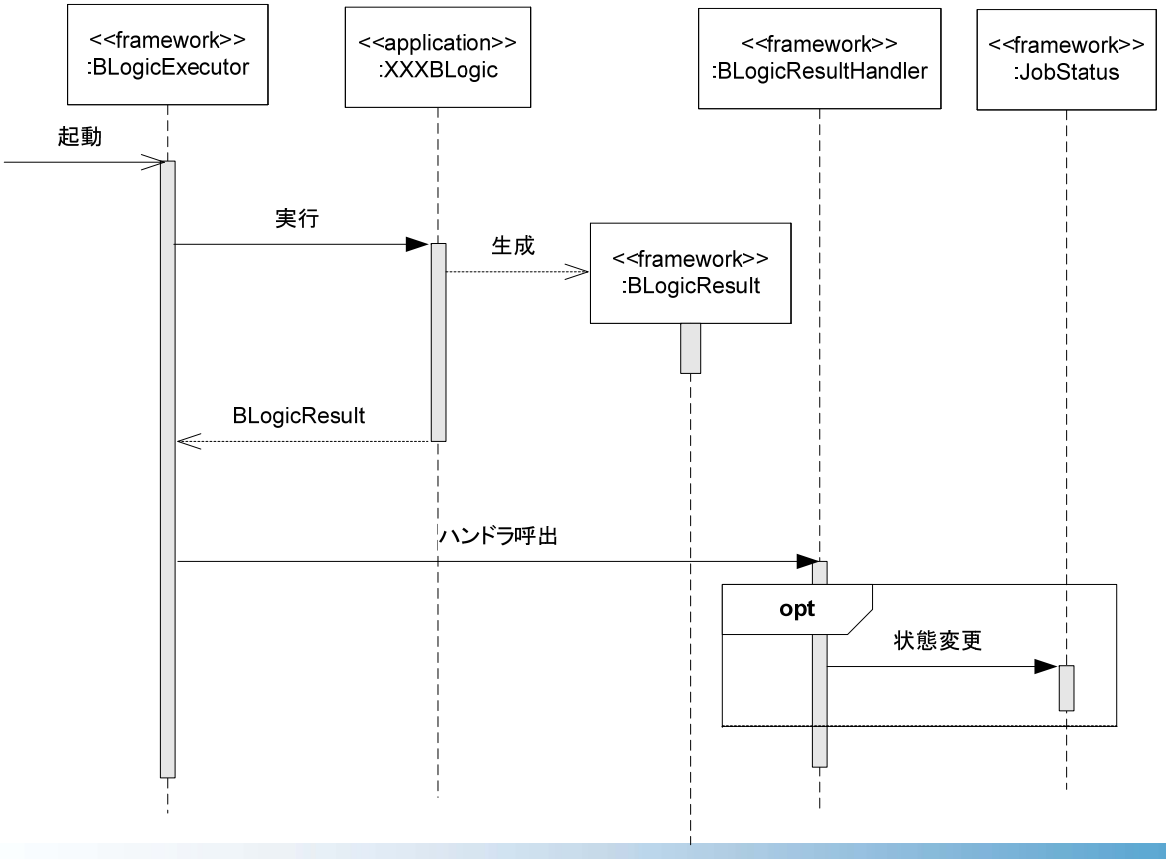
- 機能概要
  - ジョブを構成する処理（ビジネスロジックやジョブ前処理など）の処理結果を、ジョブ全体の処理結果に反映する機能である。処理結果ハンドラインタフェースならびに実装クラスを提供する。
    - たとえば、個々の処理の処理結果がエラーであった場合に、そのエラー結果をジョブ全体の処理結果に反映することでジョブを終了する。

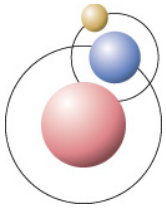
	処理結果	インタフェース 実装クラス
1	ビジネスロジック処理結果	jp.terasoluna.fw.batch.core.BLogicResultHandler
		jp.terasoluna.fw.batch.standard.StandardBLogicResultHandler jp.terasoluna.fw.batch.springsupport.transaction.TransactionalBLogicResultHandler
2	対象データ取得処理結果	jp.terasoluna.fw.batch.core.CollectorResultHandler
		jp.terasoluna.fw.batch.standard.StandardCollectorResultHandler
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

- 必要に応じて実装クラスを拡張する、もしくは置き換えることで、処理を変更することができる。（ソフトウェアアーキテクトが行なう）
- 開発者は、通常は処理結果ハンドリングに関するコーディング・設定を行なう必要はない。

BE-05 処理結果ハンドリング機能

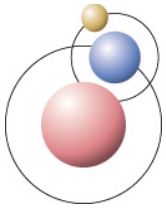
- 概念図
  - 対象とする処理(ビジネスロジック等)の処理結果は、その処理結果に対応する処理結果ハンドラ(ビジネスロジック処理結果ハンドラ等)によって処理される。





BE-05 処理結果ハンドリング機能

- 開発上の留意点
  - 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。
    - BLogicResultのリターンコードが”ERROR_CONTINUE”である場合も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。
    - BLogicResultには、フレームワークへ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。デフォルト実装では、BLogicResultに登録されたバッチ更新情報を取得し、バッチ更新リストに追加する。
    - BLogicResultのリターンコードが”ERROR_CONTINUE”または“ERROR_END”である場合、エラーデータのログ出力を行う。
  - 対象データ取得処理結果ハンドラの処理
    - CollectorResultのリターンコードが”ERROR_CONTINUE”または“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。
    - BLogicResultのリターンコードが”ERROR_CONTINUE”である場合も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。



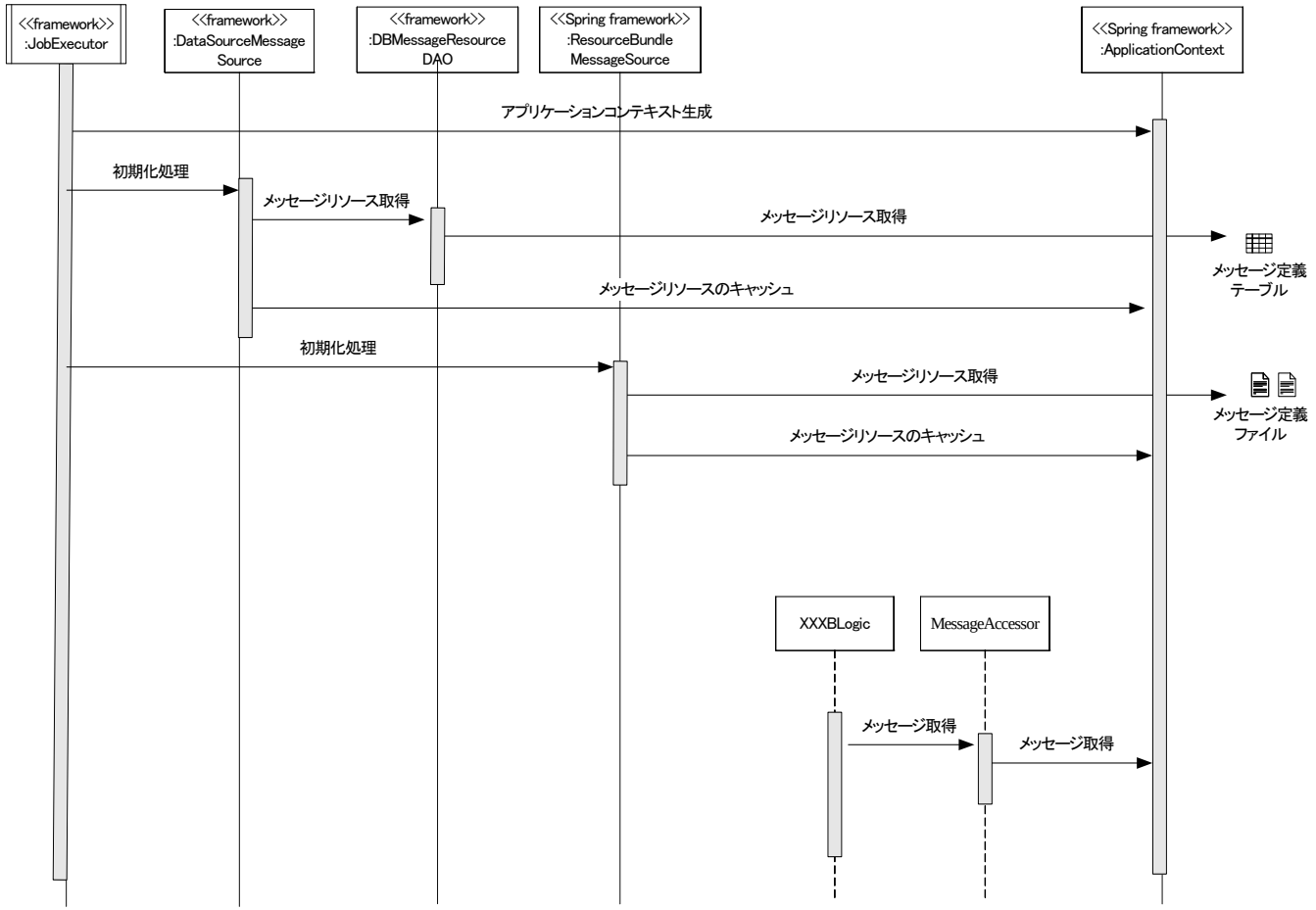
BF-01 メッセージ管理機能

- 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。
  - 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。
  - 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。

	インタフェース/クラス	説明
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	メッセージ取得の実装クラス

BF-01 メッセージ管理機能

- 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



BF-01 メッセージ管理機能

- 開発上の留意点
  - 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}
```

    - システムメッセージリソース定義ファイル（system-message.properties）

      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。

  - DBメッセージリソース
    - メッセージ定義テーブル（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。

BF-01 メッセージ管理機能

- コーディングポイント

「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.....service.XXXBlogic"/>
              <property name = "messageAccessor" ref=" messageAccessor" />
</bean>
```

メッセージ取得用実装クラスの定義

BLogicにて、メッセージ取得用実装クラスのBean定義をDIする

「ビジネスロジック」の実装例

```
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;
       }

      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

BH-01 例外ハンドリング機能

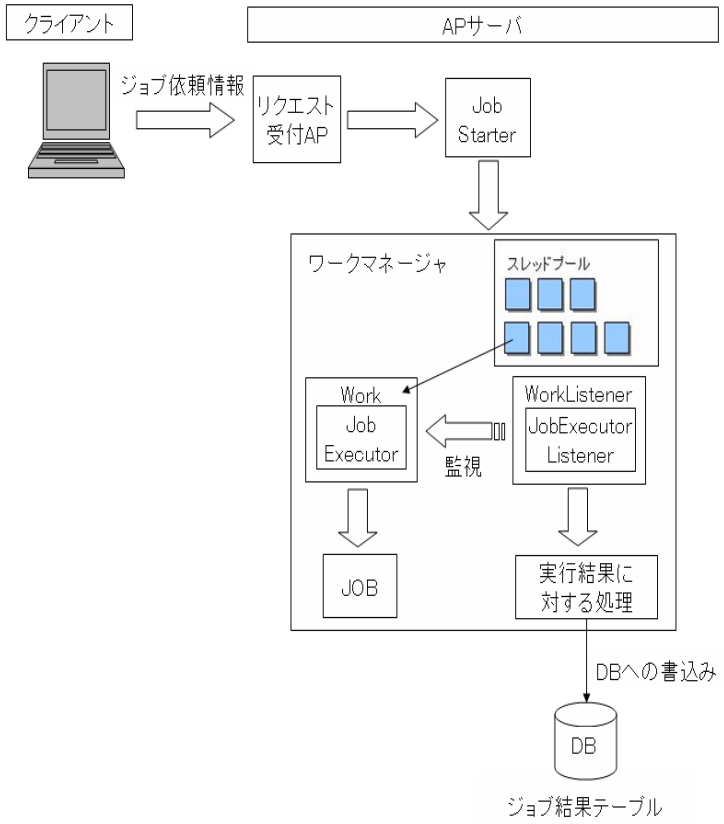
- 機能概要
  - ビジネスロジック、ジョブ前処理／ジョブ後処理などのアプリケーションで例外が発生した場合に、例外ハンドラを起動する機能を提供する。例外ハンドライインタフェースならびにデフォルト実装を提供する

	インタフェース	説明
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

- 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。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。
  - 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
- 必要に応じてデフォルト実装を拡張する、もしくは置き換えることで、処理を変更することができる。（ソフトウェアアーキテクトが行なう）
- 開発者は、必要に応じてジョブBean定義ファイルにジョブ固有の例外ハンドラマップを設定することができるが、通常は例外ハンドラに関する設定を行なう必要は無い。

BI-01 Commonj対応機能（非推奨機能）

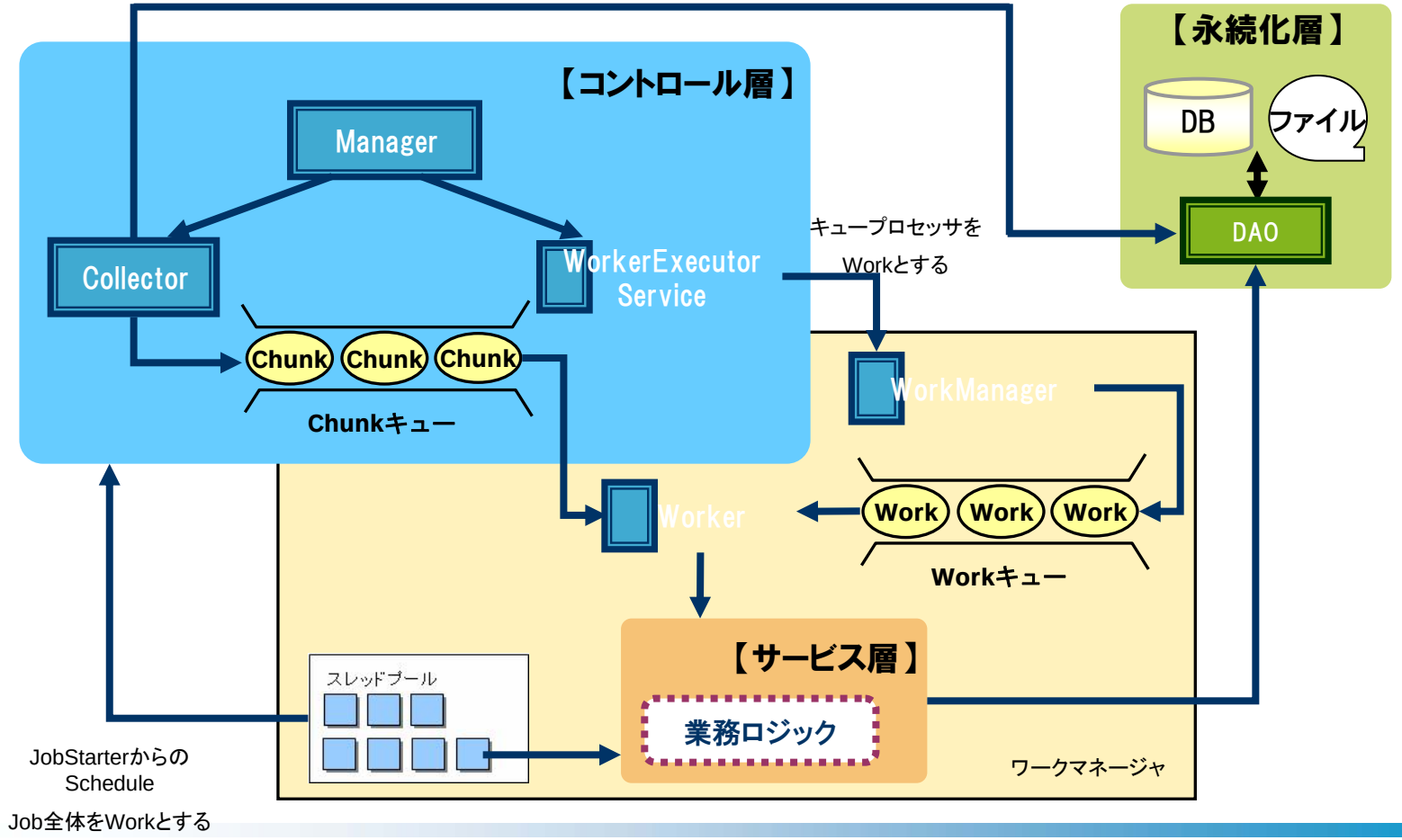
- 機能概要
  - ワークマネージャを使用したスレッド管理機能を提供する。
    - 詳細は「JSR:237 Work Manager for Application Server」を参照のこと
  - タイマーマネージャを使用したスケジューリング機能を提供する。
    - 詳細は「JSR:236 Timer for Application Server」を参照のこと。
  - クライアントから渡されてきたジョブ依頼情報を元に、APサーバ上で対象ジョブが実行される。
  - ワークマネージャがデータをチャンクキューへ詰め込み、スレッドプールからスレッドを割り当て、対象ジョブを実行する。
  - 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。



※Commonj対応機能についてはTERASOLUNA Batch Framework for Java 2.0.2.0より非推奨機能となります。
APサーバ上でジョブを動作させたい場合は「 BE-02 非同期型ジョブ起動機能」をご利用ください

BI-01 Commonj対応機能(非推奨機能)

- アーキテクチャー概観



BI-01 Commonj対応機能（非推奨機能）

- コーディングポイント

「チャンクキュー」の設定例（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

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

「ビジネスロジック」の実装例（WEB版）

WEB版BLogicからバッチを起動するときの実装例

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

}
```

引数：ジョブID、ジョブBena定義ファイル、起動パラメータを渡すことでバッチが起動される

BI-01 Commonj対応機能（非推奨機能）

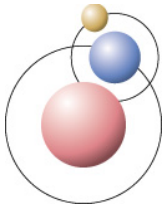
- コーディングポイント

「web.xml」の設定例

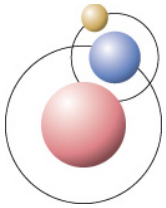
```
・・・省略・・・
<resource-ref>
    <res-ref-name>wm/BatchWorkManager</res-ref-name>
    <res-type>commonj.work.WorkManager</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>

<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>
・・・省略・・・
```

WorkManagerの設定

TimerManagerの設定

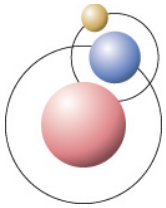


第三章 拡張ポイント説明



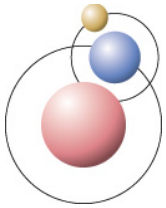
主な拡張ポイント

- ログ出力の変更
  - プロジェクト固有の要件に応じて、ログの出力タイミング、出力フォーマットなどを修正することができる。
- エラー判定の変更
  - 対象データ0件のとき、あるいは入力ファイルなし等の場合に、エラーとするか正常終了（処理なし）とするかを修正することができる。
- デフォルトのジョブ終了コードの変更
  - ビジネスロジックでジョブ終了コードを明示的に指定しなかった場合に、ジョブの成功・失敗などに応じてプロジェクトで規定したジョブ終了コードを返すようにすることができる。



拡張方法

- デフォルト実装の置き換え
- フレームワーク提供クラスの拡張
  - 継承による拡張
  - 委譲による拡張



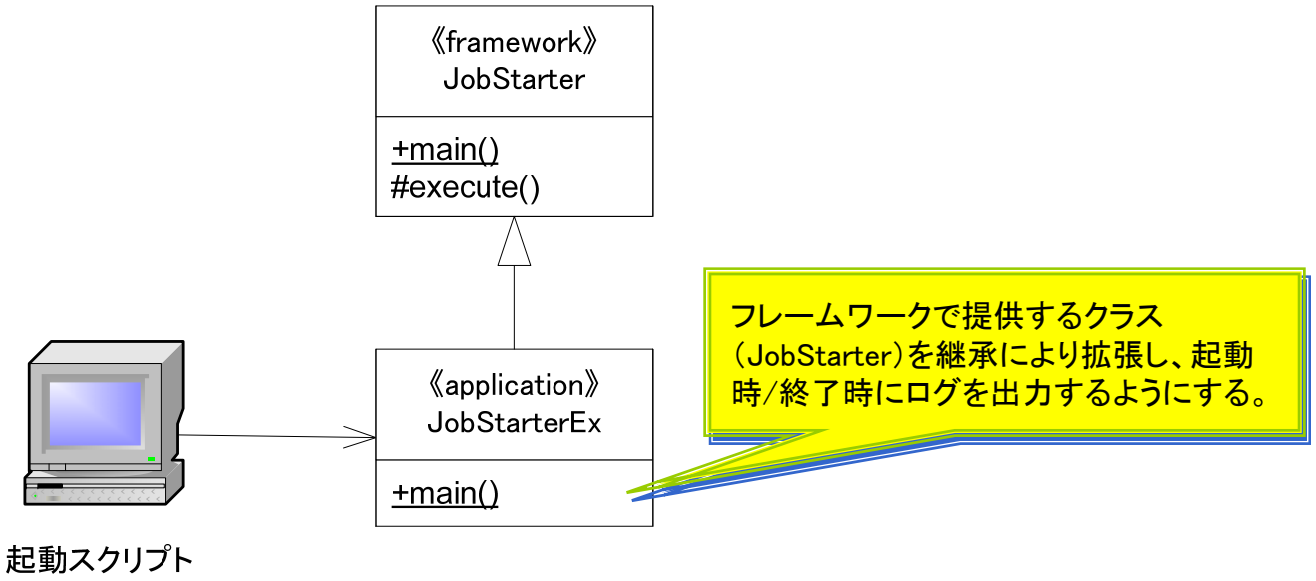
デフォルト実装の置き換え

- TERASOLUNA-Batchでは、想定される拡張ポイントにおいてインタフェースを規定し、デフォルト実装を提供している。
- デフォルト実装をプロジェクトで置き換えることによって、プロジェクト固有の要件に応じた拡張を行うことができる。

	拡張ポイント	説明
1	処理結果ハンドラ	ビジネスロジックの処理結果に応じて処理行うハンドラ。
2	例外処理ハンドラ	ジョブ実行中に発生した例外に応じて処理を行うハンドラ。
3	メッセージ取得用SQL	メッセージをデータベースから取得する際のSQL。
4	ジョブ管理テーブルアクセス用SQL	ジョブ管理テーブル（実行対象のジョブを管理するテーブル）にアクセスするためのSQL。
5	リスタート管理テーブルアクセス用SQL	リスタート管理テーブル（ジョブのリスタート情報を管理するテーブル）にアクセスするためのSQL。

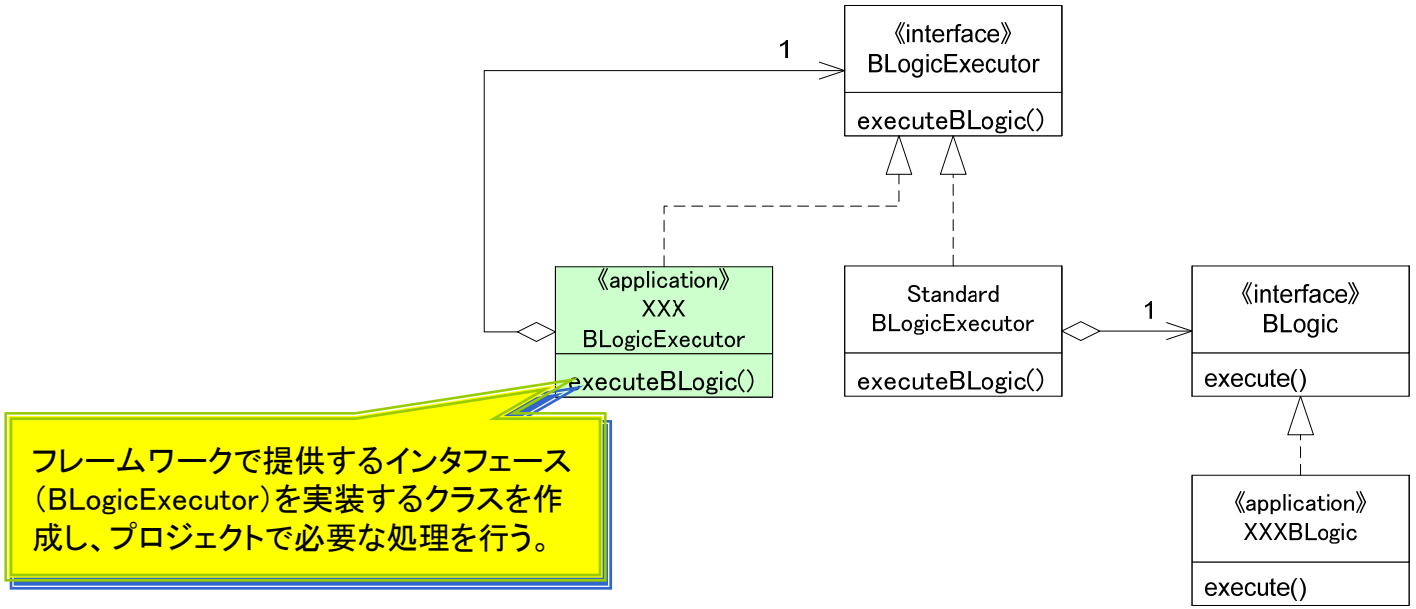
継承によるフレームワーク提供クラスの拡張

- 継承による拡張
  - フレームワークで提供するクラスを拡張し、拡張したクラスを利用するようにBean定義ファイルを修正する。
  - ジョブBean定義雛形ファイルに登録されているクラスを拡張する際には、ジョブBean定義雛形ファイルを追加/修正する。



委譲によるフレームワーク提供クラスの拡張

- 委譲による拡張
  - フレームワークで提供するインタフェースを実装するクラスを作成し、実装したクラスからフレームワークで提供するクラスに処理を委譲するようにBean定義ファイルを修正する。
  - ジョブBean定義雛形ファイルに登録されているクラスを拡張する際には、ジョブBean定義雛形ファイルを追加/修正する。



継承・委譲による拡張時のBean定義ファイル雛形の追加・修正

- 継承・委譲により拡張を行う際には、Bean定義ファイルの追加・修正を行う。

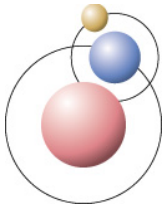
「ジョブBean定義雛形ファイル」の修正例

```
…
<bean id="jobWorker"  class="jp.terasoluna....JobWorker" >
   <property name="blogicExecutor" ref="blogicExecutor" />
     …
</bean>

<bean id="blogicExecutor" class="jp.terasoluna....XXXBLogicExecutor">
   <property name="blogicExecutorDelegate" ref="blogicExecutorDelegate" />
    …
</bean>

<bean id="blogicExecutorDelegate" class="jp.terasoluna....StandardBLogicExecutor">
   <property name="blogic" ref="blogic" />
    …
</bean>
…
```

拡張を行ったクラスを指すようにする。

追加部分。委譲による拡張を行ったクラスを登録する。

拡張したクラスから、フレームワークのクラスへ処理を委譲する。



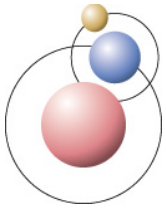
複雑なトランザクションへの対応

- フレームワークで用意されているトランザクションモデルでは対応できないような複雑なトランザクションが必要である場合には、ビジネスロジックの処理ロジックでコミット・ロールバックを行うこともできる。
- ビジネスロジックの処理ロジックでコミット・ロールバックを行う際には、ビジネスロジックにトランザクションマネージャをDIする。
  - トランザクションマネージャを呼び出すことで、コミット・ロールバックを行うことができる。
- ビジネスロジックの処理ロジックでトランザクション処理を行うと、トランザクション範囲が見えにくくなる。発見しにくいバグを生みやすいので注意すること。

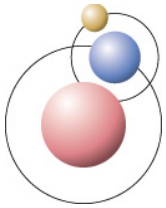
「ジョブBean定義雛形ファイル」の修正例

トランザクションマネージャをビジネスロジックにDIする。

```
<bean id="blogic" class="jp.terasoluna....XXXBLogic">
    <property name="transactionManager" ref="transactionManager" />
    ...
</bean>
```

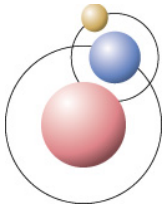


第四章 用語説明



用語説明

- 分割ジョブ
  - フレームワークが起動するジョブの種類の一つ。対象データを分割するための分割キー（都道府県コードなど）に従って、ジョブの内部の処理が多重化されているジョブ。
- 通常ジョブ
  - ジョブ内部が多重化されていないジョブ。分割ジョブではないジョブ。
- 親ジョブ
  - 分割ジョブにおいて、対象データの分割を行うジョブ。対象データの分割のみを行い、そのデータに対する処理は行わない。
- 子ジョブ
  - 分割ジョブにおいて、分割された対象データの処理を行うジョブ。
- フレームワークBean定義ファイル
  - フレームワークのJava Beanが登録されているSpringのBean定義ファイル。原則として、システムで一つ用意する。
- ジョブBean定義ファイル
  - どのように対象データを取得して、どのビジネスロジックを実行するか、などのジョブ固有の情報を定義するSpringのBean定義ファイル。
- ジョブBean
  - ジョブBean定義ファイルにおいて、ジョブ固有の情報を定義するためのJava Bean。ジョブBeanのIDは、ジョブIDでなければならない。
- ベースジョブBean
  - ジョブBeanを定義する際に雛形となるJava Bean。フレームワークBean定義ファイルに定義されており、ジョブBeanを定義する際に、parentとして指定する。



用語説明（続き）

- Collector
  - 対象データを取得するためのモジュール。フレームワークでデフォルト実装が用意されている。
- 対象データ
  - ジョブの処理対象とするデータ。データベース、あるいはファイルから取得する。
- Worker
  - 取得された対象データを処理するモジュール。フレームワークで用意されている。Workerは、アプリケーションのビジネスロジックを起動し、処理フロー制御などを行う。
- ビジネスロジック
  - 対象データの1件1件に対して行う業務処理。BLogicインタフェースを実装して、業務開発者が作成する。
- ジョブコンテキスト
  - ビジネスロジックや各種前処理などで、ジョブの実行を通して共有されるコンテキスト情報。JobContext基本クラスを継承して、業務開発者が拡張することができる。
- ジョブ終了コード
  - ジョブ終了時に、起動元のShell等に返却される終了コード。同期型ジョブ起動の場合には、Java VMの終了コードとなる。