TERASOLUNA Batch Framework for Java

1. TERASOLUNA Batch Framework for Java とは

TERASOLUNA Batch Framework for Java（以降 TERASOLUNA（Batch 版））とは、NTT データによって開発されたオープンソースの Java バッチアプリケーションフレームワークです。TERASOLUNA（Batch 版）はパーシステンス層に Apache iBATIS、DI/AOP コンテナとして Spring Framework を用いています。

ほとんどのエンタープライズアプリケーションでは、1 件 1 件同期処理を行うオンライン処理とともに、非同期で大量（バルク）処理をするバッチ処理を必要とします。これまでの開発では言語の特性により、バッチは COBOL、C、シェルスクリプト、ストアドプロシージャなどを使用して開発されることが多かったと思います。Java は Java VM 上で動くためネイティブアプリケーションに比べて、パフォーマンスが出ないという理由からオンライン処理のみ使用されてきました。しかし、以下の理由により Java でバッチ処理を行うニーズが増えてきました。

- ✧ オンライン処理と別の言語で開発することによるビジネスロジックの二重管理
- ✧ Java の性能向上およびハードウェアの性能向上
- ✧ Java エンジニアの増加

TERASOLUNA（Batch 版）はそのニーズに応え、バッチを開発するために便利な機能を提供するフレームワークです。

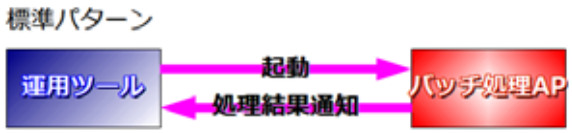
2. バッチ処理のパターン

バッチ処理には以下の 4 つのパターンがあります。

- ✧ 標準パターン
- ✧ 定期起動パターン
- ✧ イベントトリガーパターン
- ✧ オンライン非同期起動パターン

TERASOLUNA（Batch 版）では、この 4 つのパターンに対応可能です。

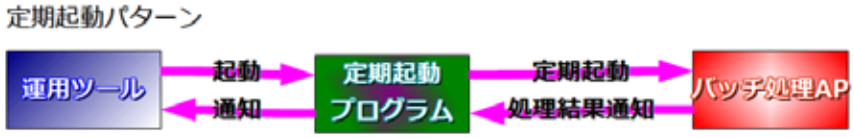
2.1. 標準パターン



基本的なバッチ処理のパターンです。バッチ処理 AP を起動する運用ツールとして使われるものは UNIX のシェルスクリプトや日立製の総合運用管理ツール JP1 などがあります。主な役割はバッチ処理 AP を起動することと、その処理結果を受け取ることです。

人が運用ツールを起動して、そこからバッチ処理を起動することになります。

2.2. 定期起動パターン



標準パターンにスケジュール駆動の機能を付け加えたものになります。標準パターンのように人手を介してバッチ処理を起動するより、スケジューラーによって定刻駆動されるこちらのパターンのほうが一般的でしょう。

夜間バッチや月次処理など、定刻に駆動する必要がある処理などがこのパターンによって実行されます。

2.3. イベントトリガーパターン

運用管理ツールによって起動されたファイル監視プログラムが特定のフォルダにファイルが到着したことを感知し、それがトリガーとなってバッチ処理が行われるパターンです。ファイルではなく、メールになっている場合やデータベースのあるテーブルにレコードがインサートされたことがトリガーとなっている場合もあります。

会計システムが月次で販売管理システムのデータを取り込んで会計データを作成する場合、データベースを介してではなく、ファイルを介してデータを同期する時があります。そのような時はこのパターンによって実行されます。

2.4. オンライン非同期起動

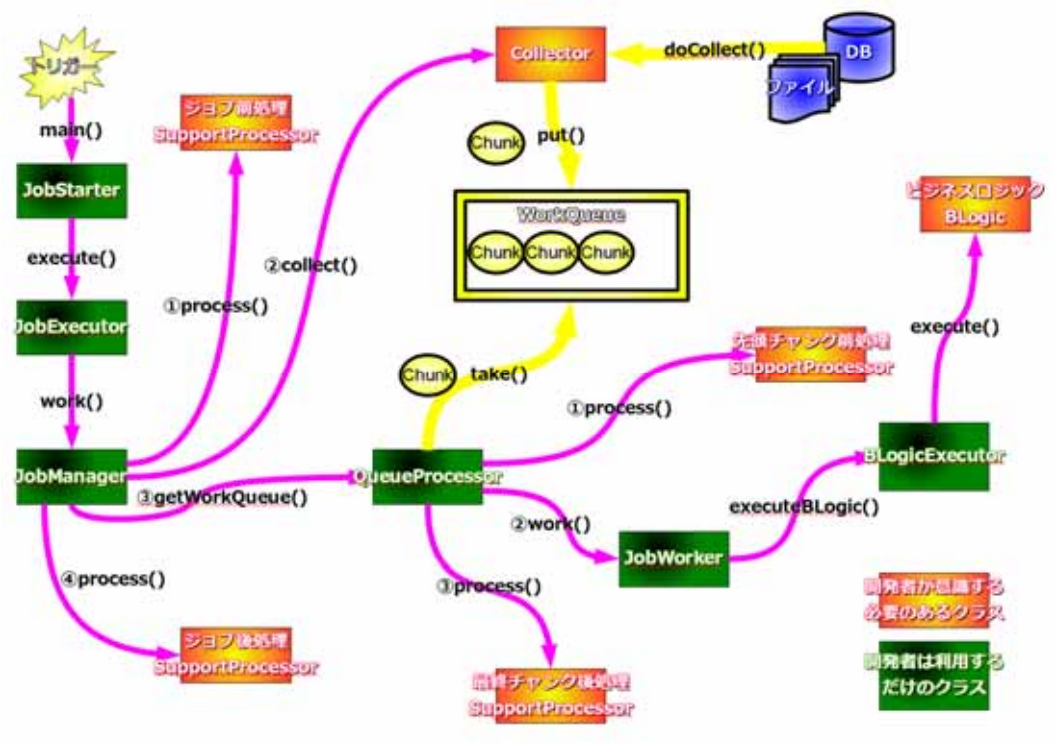
オンラインアプリケーションでクライアントからの操作がトリガーとなって、バッチ処理が起動されるパターンです。

『グーグルはページの反応が 0.5 秒遅くなるとアクセス数が 20%減るといい、アマゾンはページの反応が 0.1 秒遅くなると売り上げが 1%減るというくらい、Web サイトが表示される速度は重要なものです。』と言われるように、オンラインシステムの応答速度は重要なものです。そのため、ユーザーに即時に返答しなければならない処理と、多少時間を置いて返答しても問題ない処理を意識する必要があります。例えば、オンラインショッピングでは注文を受け付けたことは即座にユーザーに返す必要があります。しかし、その注文確認のメールは即座にユーザーに返さなければならない情報ではありません。そのような注文確認メール送信のシステムはオンラ

イン非同期処理としてバッチで実装されます。そのため、たいていのオンラインショッピングでは注文確認メールが遅延することを注意事項に示しています。そのような処理などがこのパターンによって実行されます。

3. TERASOLUNA（Batch 版）のバッチ処理

TERASOLUNA（Batch 版）のバッチ処理の全体的な流れは以下の図となります。



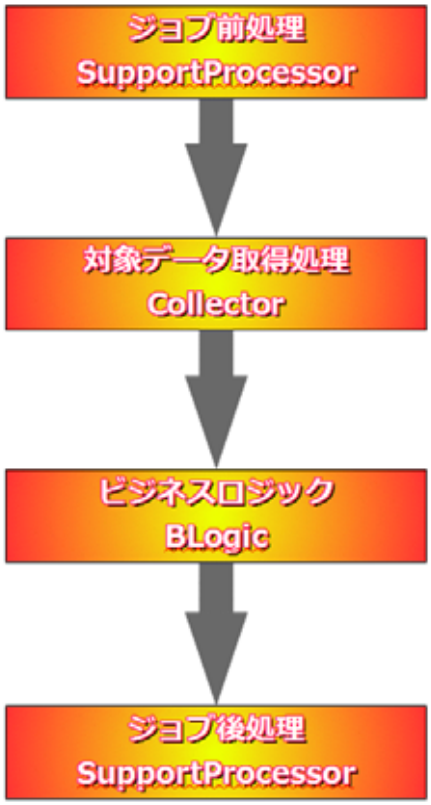
JobManager が実行するジョブの処理は以下のようなパターンがあります。

3.1. 基本形

対象データを取得し、取得した対象データを処理するビジネスロジックを実行する流れが基本形になります。

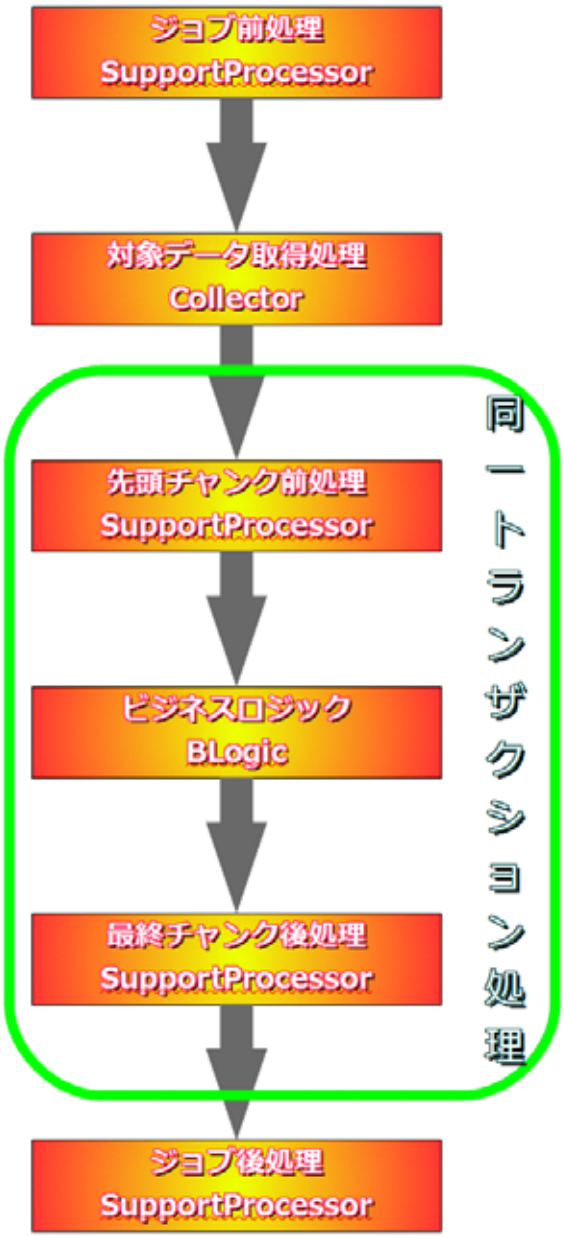
3.2. 前後処理

基本形に前処理と後処理を加えたパターンです。



3.3. 分割処理

3.2 の処理に加え、対象データをチャンクという単位に分割し、分割したデータ毎にビジネスロジックを実行できるようにしたパターンです。ビジネスロジックは同一のトランザクションで実行されます。



4. TERASOLUNA（Batch 版）の特徴

TERASOLUNA（Batch 版）は、実行処理・オンライン連携・入出力に特徴があります。

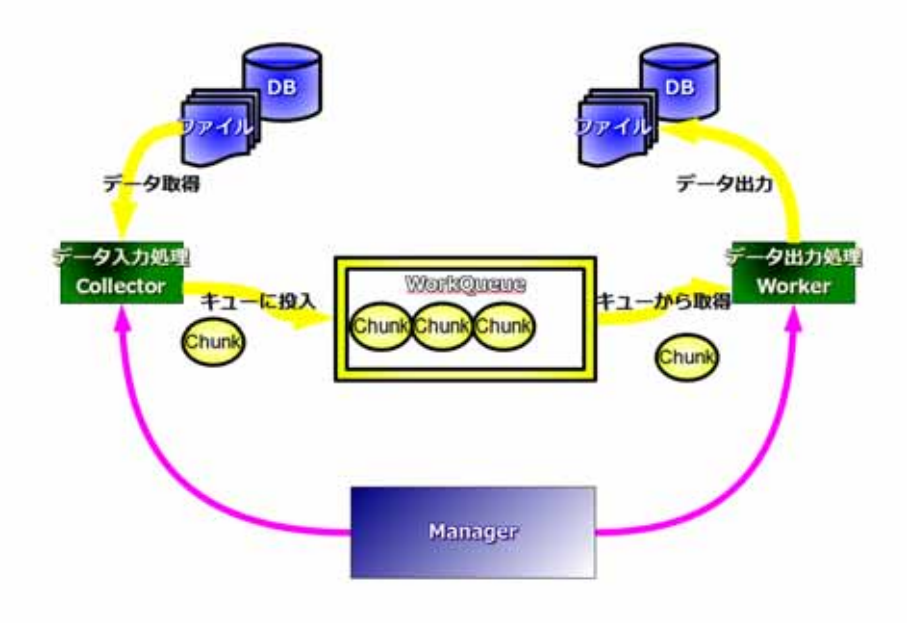
4.1. 高スループットを実現するための入力と出力の分離

バッチ処理の性能指標は『時間当たり何件の処理が実行できたか』というスループットです。TERASOLUNA（Batch 版）では高スループットを実現するために入力処理と実行・出力処理を Producer-Consumer パターンで実装しています。

- Producer-Consumer パターン

  Producer-Consumer パターンとは、以下の 3 つに分けて処理するパターンです。

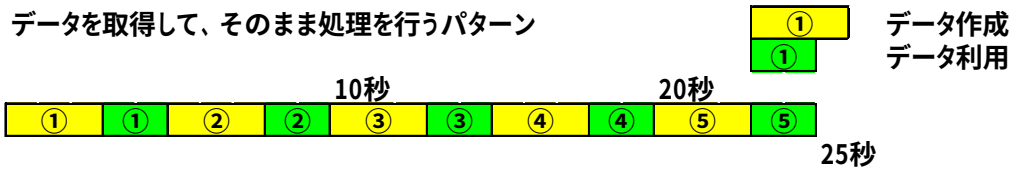
  - Producer（Collector）：処理対象のデータをチャンクに分割して、キューに投入します。
  - Consumer（Worker）：キューに保管されているチャンクを取得し、処理を行います。Collector とは別のスレッドで動作します。
  - Channel（キュー）：Collector からチャンクを取得し、保管します。また、Worker から求めに応じてチャンクを渡します。



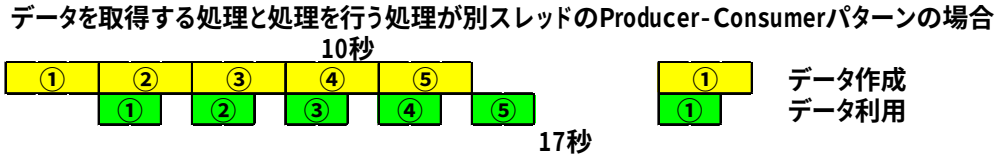
Producer-Consumer パターンで処理を行うと高スループットが実現できる理由について例を挙げて説明します。

データ 1 件当たりデータ作成に 3 秒、データ利用に 2 秒かかる処理を 5 件処理するとします。

Producer-Consumer パターンを利用せずに、データを作成してそのままデータ利用した場合、データを作成して利用が完了するまで、次のデータの作成ができません。そのため、全ての処理を終えるのに、
5 秒 x 5 件= 25 秒
かかることになります。



Producer-Consumer パターンを利用した場合、Producer のスレッドは作成したデータを Channel に渡したらすぐに次のデータ作成をします。Consumer スレッドは Channel にデータが登録されたら、すぐにそのデータを利用することができます。従ってこのパターンで全ての処理を終えるのに、
3 秒 x 5 件 + 2 秒 = 17 秒
となります。



Producer-Consumer パターンを利用することによって 8 秒の時間短縮することができます。

4.2. トランザクションコード不要

バッチ処理でデータベースに更新する時、どこからどこまでがトランザクションの範囲で、それが失敗した場合はロールバックの処理を、成功した場合はコミットの処理などを記述していきます。そのトランザクション処理はバッチ処理での要です。しかし、バッチ処理はオンライン処理に比べ、複雑なトランザクション処理になる傾向が高く、トランザクションコードが煩雑になりやすいため、コミットミスやロールバックミスを犯すリスクがあります。しかし、TERASOLUNA（Batch 版）では SpringAOP および iBATIS を利用しているため、開発者がコミットやロールバックといったトランザクションコードを記述する必要がありません。そのため、経験の浅い開発者でもトランザクションのミスせずに記述することができ、システムの

安全性が高まります。

4.3. トランザクションモデル

トランザクションには以下の 3 つのモデルがフレームワークで提供されます。

トランザクションモデル	説明
チャンク別トランザクションモデル	チャンク単位でトランザクションを張って処理を行います。チャンク単位を 1 件にすれば、1 件 1 件コミットさせることも可能です。
全チャンク単一トランザクションモデル	全てのチャンクを単一のトランザクションとして処理を行います。同じトランザクションで処理できる先頭チャンク前処理および最終チャンク後処理を設定することができます。
非トランザクションモデル	ファイルアクセスなどトランザクション管理を必要としない場合このモデルで処理を行います。

これらの用意されたトランザクションモデルでは実現できない処理は、トランザクション処理を手動で記述することができます。

4.4. チャンク単位でのコミットが可能

Web アプリケーションとは異なり、バッチアプリケーションは一度に大量のデータを扱います。その際、1 件 1 件コミットしていたらパフォーマンスが低下するため、iBATIS の JDBC バッチ更新機能を用いてチャンク単位でコミットできるようになっています。

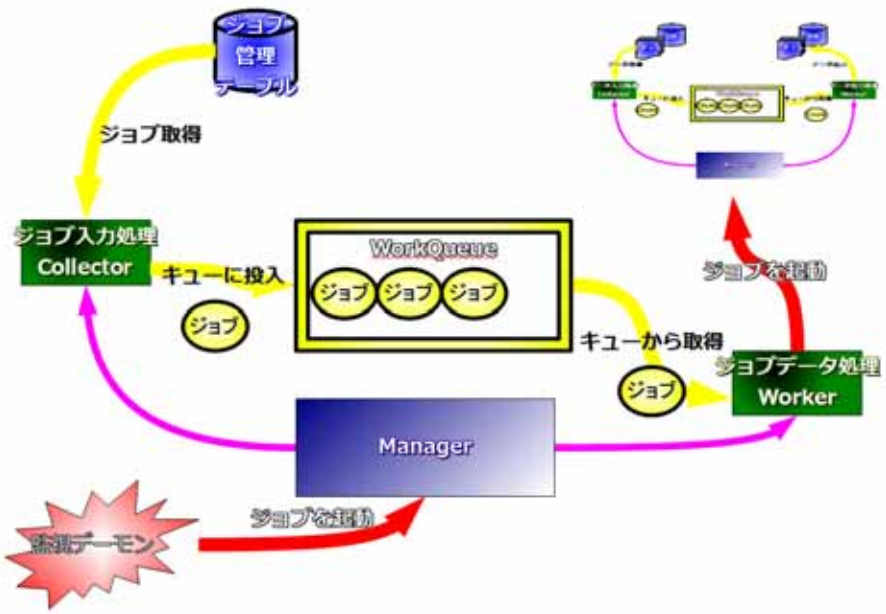
以下に実装例を示します。

```
// バッチ更新情報を格納するマップを作成する
LinkedHashMap batchUpdateMap = new LinkedHashMap();

// JDBC バッチ更新の SQL とパラメータを登録する
batchUpdateMap.put(“insertSqlA”, objectA);
batchUpdateMap.put(“updateSqlB”, objectB);
batchUpdateMap.put(“deleteSqlB”, objectC);

// BLogicResult 生成時にバッチ更新情報を格納したマップを渡す
```

```
return new BLogicResult(ReturnCode.NORMAL_CONTINUE, batchUpdateMap);
```

4.5. 監視デーモンを使用した非同期型ジョブ起動機能

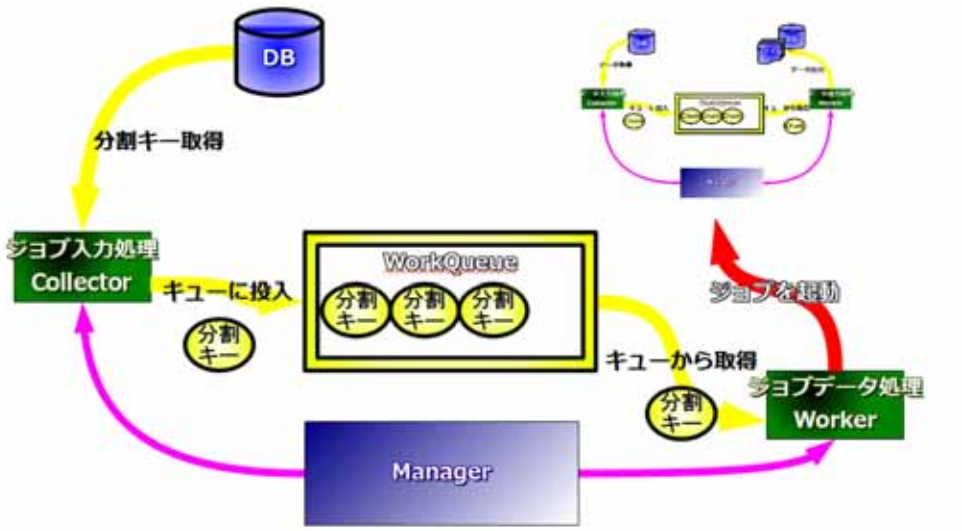
運用管理ツールなどによって直接起動される通常の同期型ジョブの他に非同期型ジョブを起動することができます。非同期型ジョブは、ジョブ管理テーブルを監視するデーモンによって起動されるジョブです。



非同期の監視デーモンが起動されます。この監視デーモンはジョブ状態管理テーブルを監視します。ジョブ状態管理テーブルにジョブレコードが登録されると、監視デーモンはそのジョブレコードを取得し、ジョブを実行します。ジョブの実行が終了すると、ジョブ終了コードをジョブ状態管理テーブルに更新します。取得したバッチ停止用レコードであれば、この非同期の監視デーモンを終了します。

4.6. 分割ジョブによる多重実行機能

入力データが分割キーにより、分割可能な時、ジョブの多重実行を行うことができる。ジョブの多重実行は並列して処理されるため、高速に処理を行うことができます。

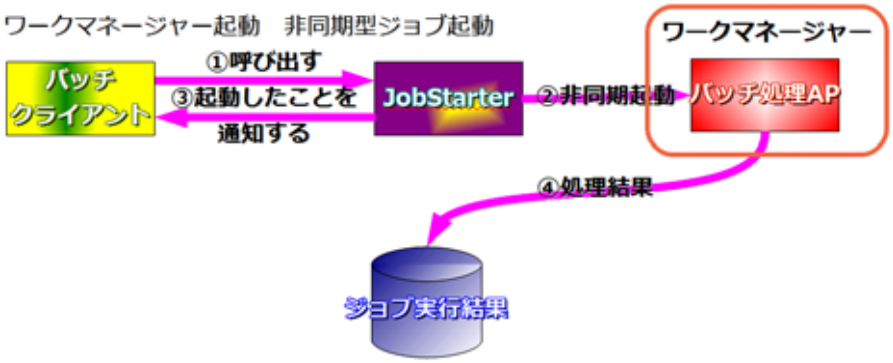


運用管理ツールなどによって、ジョブが実行されます。通常ジョブの場合、チャンク単位でデータを取得しますが、分割ジョブの場合、分割キーを取得します。取得した分割キーをキューに登録します。Worker がキューから分割キーを取り出し、それを元に新たなジョブを起動します。そのジョブの中では通常ジョブのように、チャンク単位でデータが取得され、処理されます。分割ジョブの場合は、並列して子ジョブが生成され処理されるため、非常に高速に処理を完了することができます。

4.7. オンラインサーバー上でのジョブ起動

アプリケーションサーバー上でのジョブ起動には以下の３つの方法があります。

✧　ワークマネジャーを使用した非同期型ジョブ起動



バッチ処理の処理結果を待たずにクライアントに起動通知だけを返すパターン

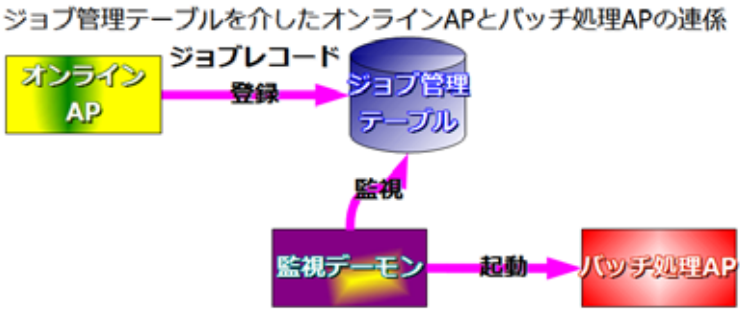
です。

✧　ワークマネジャーを使用した同期型ジョブ起動

バッチ処理の処理結果を待ってにクライアントに処理結果を返すパターンです。

✧　監視デーモンを使用した非同期型ジョブ起動

3.5 で述べた監視デーモンを使用した非同期バッチをオンラインと連携させることができます。



オンラインアプリケーションにてジョブ管理テーブルにジョブレコードをインサートし、それを監視デーモンに検知させて、オンライン非同期処理を実行することができます。

4.8. コーディングレスのファイル ⇔ JavaBean 変換

バッチ処理ではオンライン処理と違い、ファイルに対しての読み書き処理はかなり多くなる傾向があります。ファイルを開き、読み書きを行い、ファイルを閉じるといったコーディングはかなり煩雑になります。TERASOLUNA（Batch 版）ではファイルと JavaBean との変換に際し、アノテーションを用いてマッピングを行います。アノテーションには以下の 2 種類があります。

✧　ファイル全体にかかるアノテーション
✧　ファイル項目に関するアノテーション

ファイル全体に関わる設定は、DTO クラス名にアノテーションを設定します。ファイル項目に関する設定は、その DTO クラスの各フィールドにアノテーションを設定

します。以上のアノテーション設定によって、ファイル ⇔ DTO の相互変換はコーディングレスで可能です。

- ファイル全体に関わる定義情報（FileFormat アノテーション）

項番	物理項目名	論理項目名	説明	デフォルト値
1	lineFeedChar	行区切り文字	行区切り文字の設定	システムデフォルト
2	delimiter	区切り文字	区切り文字のの設定	『,（カンマ）』
3	encloseChar	囲み文字	カラムの文字列の囲み文字の設定	なし
4	fileEncoding	ファイルエンコーディング	入出力を行うファイルの文字コードを設定する	システムデフォルト
5	headerLineCount	ヘッダ行数	入力ファイルのヘッダ部の行数を設定	0
6	trailerLineCount	トレイラ行数	入力ファイルのトレイラ部の行数を設定	0
7	overWriteFlg	ファイル上書きフラグ	同じファイル名が存在した時に上書きしてよいか否かの設定	FALSE

※ システムデフォルトとは、バッチが動作する環境のシステムデフォルト値がそのまま使用されるため、意図しない形式になってしまうので、省略せずに指定したほうが無難です。

- ファイル項目に関わる定義情報（InputFileColumn アノテーション、OutputFileColumn アノテーション）

項番	物理項目名	論理項目名	説明	デフォルト値
1	columnIndex	カラムインデックス	対応するデータのカラム番目（0から）の設定	なし
2	columnFormat	フォーマット	カラムの型の設定	なし
3	bytes	バイト長	カラムのバイト長の設定	なし
4	paddingType	パディング種別	右寄せ/左寄せ/パディングなしの設定	NONE
5	paddingChar	パディング文字	パディングする文字の設定	なし

6	trimType	トリム種別	右寄せ/左寄せ/両側/トリムなしの設定	NONE
7	trimChar	トリム文字	トリムする文字の設定	なし
8	stringConverter	文字変換種別	大文字小文字変換等の設定	変換しない

4.9. オンライン処理とのバッチ処理の共通化

TERASOLUNA Server Framework for Java（Rich 版）と TERASOLUNA（Batch 版）間で、入力チェックやメッセージ取得処理などのモジュールや設定を共通化することができます。しかし、注意することはトランザクションの考え方が異なることを意識しなければなりません。

5. TERASOLUNA（Batch 版）の実装方法

TERASOLUNA（Batch 版）で実装に必要なソースコードは下表です。

名称	説明	必須か否か
設定ファイル(xml)		
ジョブ定義ファイル	ジョブの仕様を定義したSpring設定ファイル ジョブの設計書です	**必須**
iBATIS 設定ファイル	「SqlMapconfig」ファイル：「SqlMap」ファイルの情報を記述した設定ファイル 「SqlMap」ファイル：SQLを記述した設定ファイル	DB操作時、必須
入力チェック定義ファイル	ファイル・DBなどの入力データに対して入力チェックを定義した設定ファイル	任意
Java ロジック クラス		
ジョブ前処理	ビジネスロジックの前に行う処理のクラス ビジネスロジックとは別のトランザクションで処理される	任意
先頭チャンク前処理	業務処理を行う直前に処理されるクラス ビジネスロジックと同じ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時、任意 それ以外、設定不可
ビジネスロジック	業務処理を行うクラス	**必須**
最終チャンク後処理	業務処理を行った直後に処理されるクラス ビジネスロジックと同じ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時、任意 それ以外、設定不可
ジョブ後処理	ビジネスロジックの後に行う処理のクラス ビジネスロジックとは別のトランザクションで処理される	任意
Java Bean クラス		
ジョブコンテキストクラス	ジョブ全体を通して共有できる情報を格納するクラス	任意
業務入力クラス	業務処理の入力になるクラス	**必須**
ファイル行オブジェクトクラス	ファイルの入出力をアノテーションで設定したクラス	ファイルの入出力使用時、必須

以下に TERASOLUNA（Batch 版）に付随するチュートリアルを例にあげ、実装方法を説明します。今回使用したバージョンは、2.0.1.0 になります。

5.1. JOB01：データベースアクセス機能を用いたジョブ

✧　ジョブ定義ファイル

ジョブ定義ファイルは
①トランザクションモデルの雛型のインポート
②iBATIS 設定（SqlMapConfig）
③ジョブコンテキストの設定
④ジョブ前処理の設定
⑤コレクターの設定
⑥ビジネスロジックの設定
⑦ジョブ後処理の設定
を記述します。

batchapps¥tutorial¥UC0001¥JB0001.xml

```
省略

<!-①トランザクションモデル別 Bean 定義ファイルの雛形をインポート -->
<import resource="classpath:template/ChunkTransactionBean.xml"/>

<!-②SqlMapConfig-->
<bean id="sqlMapConfigFileName" class="java.lang.String">
  <constructor-arg value="tutorial/UC0001/UC0001_sqlMapConfig.xml"/>
</bean>

<!-③ジョブコンテキスト-->
<bean                                                    id="jobContext"
class="jp.terasoluna.batch.tutorial.uc0001.JB0001JobContext" />

<!-④ジョブ前処理-->
<util:list id="jobPreLogicList">
  <bean class="jp.terasoluna.batch.tutorial.uc0001.jb0001.DBJobPreLogic">
    <property name="queryDAO" ref="queryDAO" />
  </bean>
</util:list>

<!-⑤コレクタ定義-->
<bean id="collector" parent="IBatisDbChunkCollector">
  <property name="sql" value="UC0001.getNyukinData"/>
</bean>
```

```
<!-⑥ビジネスロジック-->
<bean                                                          id="blogic"
class="jp.terasoluna.batch.tutorial.uc0001.jb0001.DBBLogic">
  <property name="queryDAO" ref="queryDAO" />
  <property name="updateDAO" ref="updateDAO" />
  <property name="messageAccessor" ref="messageAccessor"/>
</bean>

<!-⑦ジョブ後処理-->
<util:list id="jobPostLogicList">
  <bean class="jp.terasoluna.batch.tutorial.uc0001.jb0001.DBJobPostLogic">
    <property name="updateDAO" ref="updateDAO" />
  </bean>
</util:list>

省略
```

バッチ処理はすべてこのジョブ定義ファイルからたどることができます。

✧ ①トランザクションモデル

この設定からデータベースのアクセスの設定をたどることができます。
batchapps¥template¥ChunkTransactionBean.xml
ではプレースホルダーの設定がされています。

```
省略
  <!-- プレースホルダ -->
  <import resource="classpath:common/PlaceHolderConfig.xml" />
省略
```

インポートしているプレースホルダー
batchapps¥ common¥PlaceHolderConfig.xml
には JDBC プロパティファイルが設定されています。

```
省略
    <!-- プレースホルダ -->
    <bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
      <property name="locations">
        <list>
          <value>classpath:jdbc.properties</value>
          <value>classpath:template/workQueueFactory.properties</value>
        </list>
      </property>
  </bean>
省略
```

JDBC プロパティファイル
batchapps¥jdbc.properties
DB 接続先が設定されています。

```
省略
# local HSQLDB
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://127.0.0.1/terasoluna
jdbc.username=sa
jdbc.password=
省略
```

✧ ②SqlMapConfig 設定ファイル

ここに記載された iBATIS の SqlMapConfig 設定ファイル
tutorial/UC0001/UC0001_sqlMapConfig.xml
には、iBATIS の SqlMap 設定ファイルへのパスが記載されています。

```
省略
<!-- サンプル用 sqlMap の定義-->
<sqlMap resource="tutorial/UC0001/UC0001_sqlMap.xml"/>
省略
```

iBATIS の SqlMap 設定ファイルには、コレクターやビジネスロジックなどで使用する SQL 文が記載されています。

```
省略
<sqlMap namespace="UC0001">

  <!-- ジョブ前処理  -->
  <select id="getUnyohiduke" resultClass="java.util.Date">
      SELECT UNYOHIDUKE FROM UNYOHIDUKETABLE
  </select>

  <!-- 対象データ取得処理 -->
  <select id="getNyukinData"
 resultClass="jp.terasoluna.batch.tutorial.uc0001.JB0001Data">
      SELECT ID, SITEN, KOKYAKUID, NYUKIN, TORIHIKIBI FROM NYUKINTABLE ORDER BY
ID
  </select>

  <!-- 対象データ取得処理（ジョブ分割） -->
  <select id="getPartitionNyukinData"
 parameterClass="jp.terasoluna.batch.tutorial.uc0001.JB0001JobContext"
```

```
      resultClass="jp.terasoluna.batch.tutorial.uc0001.JB0001Data">
      SELECT ID, SITEN, KOKYAKUID, NYUKIN, TORIHIKIBI FROM NYUKINTABLE WHERE SITEN
= #partitionKey# ORDER BY ID
  </select>

  <!-- ビジネスロジック -->
  <select id="getZandakaData" parameterClass="java.lang.String"
      resultClass="jp.terasoluna.batch.tutorial.uc0001.jb0001.ZandakaData">
      SELECT ZANDAKA FROM ZANDAKATABLE WHERE KOKYAKUID = #value#
  </select>
  <update id="updateZandakaData"
 parameterClass="jp.terasoluna.batch.tutorial.uc0001.jb0001.ZandakaData">
      UPDATE ZANDAKATABLE SET ZANDAKA = #zandaka#, SAISYUTORIHIKIBI =
#saisyutorihikibi# WHERE KOKYAKUID = #kokyakuid#
  </update>
  <insert id="insertZandakaData"
 parameterClass="jp.terasoluna.batch.tutorial.uc0001.jb0001.ZandakaData">
      INSERT INTO ZANDAKATABLE(KOKYAKUID, ZANDAKA, SAISYUTORIHIKIBI) VALUES
(#kokyakuid#, #zandaka#, #saisyutorihikibi#)
  </insert>

  <!-- ジョブ後処理 -->
  <insert id="insertZandakaRireki"
parameterClass="jp.terasoluna.batch.tutorial.uc0001.jb0001.ZandakaRirekiData">
      INSERT INTO ZANDAKARIREKITABLE(SYORIBI, SYORIKENSU) VALUES (#syoribi#,
#syorikensu#)
  </insert>

</sqlMap>
```

SQL へ渡す引数には『#』で区切ります。

✧ ③ジョブコンテキスト

バッチアプリケーションの各処理で共有したい情報を格納できます。
jp.terasoluna.batch.tutorial.uc0001.JB0001JobContext

```
省略
public class JB0001JobContext extends JobContext {
省略
/**
 * 運用日付
 */
private Date unyohiduke = null;
```

```
/**
 * 処理件数
 */
private int       = 0;
省略
```

運用日付と処理件数をバッチ処理全体で共有しようとしていることがわかります。

✧ ④ジョブ前処理

ビジネスロジックを行う前の処理です。queryDAO が設定されているため、ジョブ前処理でデータベースからデータを取得しようとしていることがわかります。ジョブコンテキストで運用日付と処理件数が設定されていたため、おそらく運用日付をデータベースから取得するのではないかと想像されます。
jp.terasoluna.batch.tutorial.uc0001.jb0001.DBJobPreLogic

```
省略
public class DBJobPreLogic implements SupportLogic<JB0001JobContext>
省略
/**
 * ジョブ前処理を実行する。
 * “運用日付テーブル”より“運用日付を”取得し、ジョブコンテキストに設定する。
 * @param jobContext ジョブコンテキスト
 * @return ビジネスロジック処理結果
 */
public BLogicResult execute(JB0001JobContext jobContext) {

    //“運用日付”を取得し、ジョブコンテキストに設定する。
    Date unyohiduke = queryDAO.executeForObject("UC0001.getUnyohiduke",
            null, Date.class);
    jobContext.setUnyohiduke(unyohiduke);

    //ビジネスロジック処理結果オブジェクトを返却する。
    return new BLogicResult(ReturnCode.NORMAL_CONTINUE);
}
省略
```

予想通り、運用日付を取得する処理がビジネスロジックのジョブ前処理としておこなわれます。

✧ ⑤コレクタ定義

ビジネスロジックで処理するデータを取得する定義です。ネームスペースが UC0001 で sqlId が getNyukinData に該当する SQL を sqlMap より取得します。

tutorial/UC0001/UC0001_sqlMap.xml

```
省略
  <!-- 対象データ取得処理 -->
  <select id="getNyukinData"
 resultClass="jp.terasoluna.batch.tutorial.uc0001.JB0001Data">
      SELECT ID, SITEN, KOKYAKUID, NYUKIN, TORIHIKIBI FROM NYUKINTABLE ORDER
BY ID
  </select>

省略
```

コーディングレスでデータベースより処理対象データを取得できます。

✧ ⑥ビジネスロジック

コレクターで取得したデータを処理するビジネスロジックです。queryDAO、updateDAO、messageAccessor が設定されているため、ビジネスロジックの処理はデータベースからデータを取得し、それを加工してデータベースを更新し、メッセージを表示していると予想できます。
jp.terasoluna.batch.tutorial.uc0001.jb0001.DBBLogic

```
省略
public class DBBLogic implements BLogic<JB0001Data, JB0001JobContext> {
省略
/**
 * ビジネスロジックを実行する。
 * @param nyukinData 入力パラメータ
 * @param jobContext ジョブコンテキスト
 * @return ビジネスロジック処理結果
 */
public BLogicResult execute(JB0001Data nyukinData,
        JB0001JobContext jobContext) {

  //取引日と運用日付の比較
  Date torihikibi = nyukinData.getTorihikibi();
  Date unyohiduke = jobContext.getUnyohiduke();


  if (!torihikibi.before(unyohiduke)) {

    //取引日 >= 運用日付 ならばメッセージを取得しログに出力する。
```

```
    String[] args = {nyukinData.getKokyakuid()};
    String message = messageAccessor.getMessage("msg.SampleMsg", args);

    log.warn(message);

    /* 「2.2.5 例外処理の実装」では以下のコメント行を有効にする。 */
    /*
    //ビジネスロジック結果オブジェクトを返却する。
    return new BLogicResult(ReturnCode.ERROR_END);

    //任意のジョブ終了コードを設定する場合は、以下のように第 2 引数にジョブ
終了コードを指定する。
    //return new BLogicResult(ReturnCode.ERROR_END, 99);
    */

  }

  //残高テーブル参照
  ZandakaData zandakaData =
    queryDAO.executeForObject("UC0001.getZandakaData",
    nyukinData.getKokyakuid() , ZandakaData.class);

  //該当レコードがある場合は残高計算と更新
  if (zandakaData != null) {
    int zandaka = zandakaData.getZandaka() + nyukinData.getNyukin();
    zandakaData.setZandaka(zandaka);
    zandakaData.setSaisyutorihikibi(nyukinData.getTorihikibi());
    zandakaData.setKokyakuid(nyukinData.getKokyakuid());
    updateDAO.execute("UC0001.updateZandakaData", zandakaData);

  //該当レコードが無い場合は新規作成
  } else {
    zandakaData = new ZandakaData();
    zandakaData.setKokyakuid(nyukinData.getKokyakuid());
    zandakaData.setZandaka(nyukinData.getNyukin());
    zandakaData.setSaisyutorihikibi(nyukinData.getTorihikibi());
    updateDAO.execute("UC0001.insertZandakaData", zandakaData);
  }

  //処理件数をカウントアップ
  jobContext.incrementCount();

  //ビジネスロジック処理結果オブジェクトを返却する
  return new BLogicResult(ReturnCode.NORMAL_CONTINUE);
}
省略
```

nyukinData はコレクターで取得したデータになります。ジョブ前処理で取得したデータと処理対象データの個々のデータを比較して、処理を行っています。バッチ処理全体で共有できるジョブコンテキストに処理件数をカウントアップしています。処理対象データが 2 件以上場合でも開発者がループ処理をする必要はなく、フレームワークが処理件数だけ自動的にループ処理を行います。

✧ ⑦ジョブ後処理

ビジネスロジックを行った後の処理です。updateDAO が設定され、ビジネスロジックでジョブコンテキストに処理件数を入れているため、その処理件数を更新・インサートするのではないかと想像できます。。
jp.terasoluna.batch.tutorial.uc0001.jb0001.DBJobPostLogic

```
省略
public class DBJobPostLogic implements SupportLogic<JB0001JobContext> {
省略
/**
 * ジョブ後処理を実行する。
 * @param jobContext ジョブコンテキスト
 * @return ビジネスロジック処理結果
 */
public BLogicResult execute(JB0001JobContext jobContext) {

  //“残高履歴テーブル”にレコードを追加する。
  Date unyohiduke = jobContext.getUnyohiduke();
  int count = jobContext.getCount();

  ZandakaRirekiData rireki = new ZandakaRirekiData();
  rireki.setSyoribi(unyohiduke);
  rireki.setSyorikensu(count);

  updateDAO.execute("UC0001.insertZandakaRireki", rireki);

  //ビジネスロジック処理結果オブジェクトを返却する。
  return new BLogicResult(ReturnCode.NORMAL_CONTINUE);
}
省略
```

予想通り、ジョブコンテキストの内容をデータベースにインサートしています。

いかがでしょうか。TERASOLUNA（Batch 版）はジョブ定義ファイルからそのすべてがたどれるようになっています。ベースとなるジョブを作成して、それから派生したジョブを作成するの容易に行えます。

5.2. JOB02：ファイルアクセス機能を用いたジョブ

ファイルアクセス機能は以下のようにクラスにアノテーションを記述するだけでファイルと JavaBean との相互変換を自動的に行うことができます。

jp.terasoluna.batch.tutorial.uc0001. JB0002Data

```
省略

/**
 * “入金データファイル”用ファイル行オブジェクト。
 */
@FileFormat(overWriteFlg = true, encloseChar = '¥"')
public class JB0002Data {

省略

/**
 * 取引 ID
 */
@InputFileColumn(columnIndex = 0)
private int id = 0;

/**
 * 支店
 */
@InputFileColumn(columnIndex = 1)
private String siten = null;

/**
 * 顧客 ID
 */
@InputFileColumn(columnIndex = 2, bytes = 3)
@OutputFileColumn(columnIndex = 0)
private String kokyakuid = null;

/**
 * 入金金額
 */
@InputFileColumn(columnIndex = 3)
@OutputFileColumn(columnIndex = 1)
private int nyukin = 0;

/**
 * 取引日
```

```
 */
@InputFileColumn(columnIndex = 4, bytes = 10, columnFormat = "yyyy-MM-dd")
@OutputFileColumn(columnIndex = 2, columnFormat = "yyyy/MM/dd")
private Date torihikibi = null;

省略
```

6. まとめ

TERASOLUNA（Batch 版）にはジョブ間の依存関係を定義し起動するジョブスケジューリング機能がありません。そのため、OS 標準の cron などのスケジューラー、オープンソースの Hinemos（NTT データ）、JP1（日立）や Systemwalker（富士通）などを用いる必要があります。

コーディングレスのファイル入出力は、TERASOLUNA（Batch 版）を使用しない場合は実装に手間がかかりますが、アノテーションで非常に簡単に実装することができます。また、Producer-Consumer パターンによる入出力の分離で、パフォーマンスを重視したバッチフレームワークになっています。特に CPU がマルチコアである場合は、分割ジョブなどによりハイパフォーマンスが期待できます。以上の点からも、TERASOLUNA（Batch 版）はとても実用的なバッチアプリケーションフレームワークと言えます。

7. 参考文献

- TERASOLUNA Batch Framework for Java　アーキテクチャ説明書
- TERASOLUNA Batch Framework for Java　機能説明書
- TERASOLUNA Batch Framework for Java　設定ファイル説明書

  以上の 3 つのドキュメントについては
  http://sourceforge.jp/projects/terasoluna/releases/
  より『terasoluna-batch4j-doc_2.0.1.0.zip』をダウンロードすることにより取得できます。

- @IT 総合トップ ＞ テクノロジー ＞ Java Solution ＞ Java バッチ処理は本当に業務で“使える”の？
  http://www.atmarkit.co.jp/fjava/index/index_javabatch.html
- 安藤幸央のランダウン[37]バッチ処理は Java でバッチリ？その現状とこれから
  http://www.atmarkit.co.jp/fjava/column/andoh/andoh37.html

- 『アプリケーション開発を成功に導く システム基盤の構築ノウハウ』
  著者：谷口 俊一、石川 辰雄、沢井 良二、鈴木 広司
  出版社：日経 BP 社
  ISBN：4-8222-2972-6
- 『増補改訂版 Java 言語で学ぶデザインパターン入門 マルチスレッド編 』
  著者：結城 浩
  出版社：ソフトバンククリエイティブ
  ISBN：4-7973-3162-3

開発部
荒川　正義