Chapter

15

Spring Batch

多くのエンタープライズシステムでは、前章までに学んだWebアプリケーションだけではなく、バッチアプリケーションの開発も行なうことが一般的です。

本章では、バッチアプリケーション開発のためのフレームワークである、Spring Batchの概要と基本的な利用方法について紹介していきます。

15.1 Spring Batchについて

本書ではこれまで、Webアプリケーションを開発する方法を中心に行なってきました。本章ではWebアプリケーションと対となるバッチアプリケーションについて、Springを活用して開発する方法を解説していきます。まず、一般的なエンタープライズシステムにおけるバッチ処理の特徴や求められる要件を確認してから、Spring Batchの概要を紹介します。次に、シンプルなSpring Batchアプリケーションを作成しながらSpring Batchの基礎を学び、最後に、Spring Batchのアーキテクチャについて理解を深めていきましょう。

15.1.1 バッチ処理とは

Spring Batchの話に入る前に、バッチ処理の特徴について整理しておきましょう。

これまでの章で紹介してきたWebアプリケーションのように、利用者からのリクエストを処理し、リアルタイムに結果を利用者へ返却する処理方式を「オンライン処理」と呼びます。「オンライン処理」は即座に応答することが最大の特徴であり、これにより、常に最新の情報を扱うことができたり、異なる利用者からの複数のリクエストを同時に処理できたりするというメリットがあります。昨今のシステムでは欠かせない処理方式となっています。

オンライン処理方式と対になるものが「バッチ処理」であり、以下のような特徴があります。

- 一定量のデータをまとめて処理する
- 処理に一定の順序がある

バッチ処理はオンライン処理と比較して、応答性よりも処理スループットを優先した処理方式です。バッチ処理という言葉で、何千万件という大量データを短時間に処理するイメージを持たれる読者の方も多いのではないのでしょうか。しかし、それ以外にも業務特性上、オンライン処理よりもバッチ処理のほうが有利であることが多いのです。

以下は、バッチ処理を利用する目的の一例です。

- **スループットの向上**
  データをまとめて処理することは、処理のスループットを向上させるのに有効に働きます。ファイルやデータベースは、処理1件ごとにデータを入出力するのではなく一定件数にまとめることで、I/O待ちのオーバーヘッドが劇的に少なくなり効率的です。1件ごとのI/O待ちは微々たるものですが、大量データを処理

する場合は致命的な遅延となってしまいます。バッチ処理によってこのような遅延を回避できるようになります。

- **応答性能の確保**
  オンライン処理の応答性を確保するために、オンライン処理で行なう必要がない部分をバッチ処理に切り出すことがあります。たとえばオンライン処理において、すぐに処理結果を得る必要がない場合、オンライン処理で受付までを処理して、裏で逐次バッチ処理で処理するような構成にします。
- **業務とシステムの考え方の一致**
  エンタープライズシステムでは、しばしば一定の時間にまたがる処理が登場します。これを素直に実装できるのがバッチ処理です。たとえば、業務要件に従って1か月間のデータの集計する、システム運用ルールに則って週末日曜の午前2時に1週間分の業務データのバックアップを取る、といったことです。
- **外部システムとの連携上の制約**
  ファイルなど外部システムとのインターフェイスが制約となって、バッチ処理が利用されることもあります。外部システムから送付されてきたファイルは、一定期間のデータのまとまりであるのが一般的です。このようなファイルを受け取りデータを順次取り込むような処理では、オンライン処理よりもバッチ処理のほうが処理の考え方が合っています。

15.1.2 バッチ処理が満たすべき要件

オンライン処理（特にWebアプリケーション）と比較すると、バッチ処理はさまざまな要素技術を組み合わせて実装することが多くなります。一定規模以上のエンタープライズシステムならば、必ずと言っていいほど登場する技術があります。前項で挙げたような目的を実現するには、これら要素技術を組み合わせて使用していきます。

- **ジョブスケジューラ**
  バッチ処理の1実行単位をジョブと呼びます。エンタープライズシステムではジョブが数百、数千以上に至る場合がよくあります。それらの関連を定義したり、実行スケジュールを管理したりする仕組みが不可欠です。ジョブスケジューラとは、それらを実現するためのミドルウェアです。
- **シェルスクリプト（bash など）**
  ジョブの実装方法の1つです。OSやミドルウェアなどに実装されているコマンドを組み合わせて1つの処理を実現します。手軽な半面、複雑なビジネスロジックを記述するには不向きであるため、ファイルコピーやバックアップ、テーブルクリアなど、主にシンプルな処理に用います。別のプログラミング言語で実装した処理を実行する際に、その起動前の設定や終了後の処理だけをシェルスクリプトが担うことも多くあります。
- **Java をはじめとするプログラミング言語**
  ジョブの実装方法の1つです。シェルスクリプトよりも構造化されたコードを記述することができ、開発生

産性、メンテナンス性、品質確保などを確保するのに有利です。そのため、比較的複雑なロジックとなりがちな、ファイルやデータベースのデータを加工する場合によく使われます。

では、バッチ処理が満たすべき要件は何でしょうか？ そして、それはどのような仕組みならば可能なのでしょうか？ もちろんシステムによってさまざまですが、代表的なものを以下に挙げます。

- **大量データを一定のリソースで効率よく処理できる**
  前述の処理スループットの話と密接に関連しますが、大量のデータをまとめて処理することで処理時間を短縮できるようにします。このとき重要なのは、「一定のリソースで」の部分です。100万件でも1億件でも、一定のCPU／メモリの使用で処理でき、件数に応じてゆるやかかつリニアに処理時間が延びるのが理想です。まとめて処理するということは一定件数ごとにトランザクションを開始／終了する仕組みが必要となり、処理時間を短縮するには一定件数をまとめて入出力することでI/Oを低減させながら、使用するリソースを平準化する仕組みが必要となります。それでも処理しきれない膨大なデータが相手になる場合には、一歩進んでハードウェアリソースを限界まで使い切る仕組みも追加で必要になります。処理対象データを件数やグループで分割して、複数プロセス、複数スレッドによって多重処理します。この限界まで使い切る際には、I/Oを限りなく低減することが極めて重要です。
- **可能な限り処理を継続できる**
  大量データを処理するにあたって、入力データが異常な場合や、システム自体に異常が発生した場合の防御策を考えておく必要があります。大量データは必然的に処理し終わるまでに長時間かかることが予想できますが、エラー発生による処理の中断の場合は復旧までの時間が予測できず、システムの運用に大きな影響を及ぼすことがあるためです。たとえば、100万件のデータを処理する場合に、99万件目でエラーになり、それまでのすべての処理をやり直すとしたら、運用スケジュールに影響が出てしまうことは明白です。このような影響を抑えるために、バッチ処理ならではの処理継続性が重要となります。これには、エラーデータをスキップしながら次のデータを処理する仕組み、可能な限り自動復旧を試みる仕組み、処理をリスタートする仕組みなどが必要となります。1つのジョブを極力シンプルなつくりにし、再実行を容易にすることも重要です。
- **実行契機に応じて柔軟に実行できる**
  時刻を契機とした場合、オンラインや外部システムとの連携を契機とした場合、さまざまな実行契機に対応するために、バッチ処理の実行には柔軟さが求められます。ジョブスケジューラから定時になったらプロセスを起動させ処理が完了したらプロセスを終了させたり、プロセスを常駐させておき随時バッチ処理を起動したりといったバリエーションが一般的に知られており、これを実現する仕組みが必要となります。
- **さまざまな入出力インターフェイスを扱える**
  オンラインや外部システムと連携するということは、データベースはもちろん、CSV/XMLといったさまざまなフォーマットのファイルを扱えることが重要となります。さらに、それぞれの入出力の形式を透過的に扱える仕組みがあると実装しやすくなり、複数フォーマットへの対応も迅速に行なえるようになります。

一般に、バッチ処理で実現されたアプリケーションをバッチアプリケーションと呼び、前述のスクリプトやプログラミング言語で実装した処理が該当します。Spring Batch は、Java によるバッチアプリケーションを開発する際のフレームワークであり、前述のようなバッチ処理に求められる要件を達成するための仕組みがいくつも備わっています。

COLUMN

バッチ処理なのに常駐？

先ほど、バッチ処理なのにプロセスを常駐させる場合について説明しました。ここで、「バッチ処理なのに常駐ってどういうこと？」と違和感を覚えた方もいるかもしれません。もう少し具体的なケースを考えてみましょう。

このような方法が必要になるケースとは、オンライン処理からの連携時などで高頻度にバッチ処理を実行する場合です。たとえば、オンライン処理を入り口として、ある業務の詳細レポートを出力する処理を考えてみましょう。 オンライン処理ですべて行なうと該当データの取得からデータの作り込み、レポート出力までを行なわなければならなくなり応答性を犠牲にしてしまいます。そのため、オンライン処理では詳細レポート出力依頼だけを受付け、バッチ処理と連携して実際にレポートを出力してユーザーに送付することで、応答性と処理性能を両立させます。実際、このような処理が高頻度で発生する場合にプロセスを逐一起動してしまうと、それだけでリソースを多く使ってしまい、処理時間も増えてしまいます。

このような処理方式は一般にバッチの一種としてグルーピングされており、「ディレイドバッチ」、「オンラインバッチ」などと呼ばれています。

15.1.3 Spring Batchとは

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

- **処理の流れを定型化する**
  - **シンプルな処理：タスクレット方式**
    SQL を 1 回発行するだけ、コマンドを発行するだけ、といったケースで用いる
  - **大量データを効率よく処理する：チャンク方式**
    データの取得／加工／出力といった処理の流れを定型化し、一部のみ実装すればよい。一定件数をひとまとめにしたトランザクション管理は Spring Batch が担う
- **さまざまな起動方法**
  コマンドライン実行、Servlet 上で実行、その他のさまざまな契機での実行を実現する
- **さまざまなデータ形式の入出力**
  ファイル、データベース、メッセージキューをはじめとするさまざまなデータリソースとの入出力を簡単に

行う

- **処理の効率化**
  多重実行、並列実行、条件分岐を設定ベースで行なう
- **ジョブの管理**
  実行状況の永続化、データ件数を基準にしたリスタートなどを可能にする

Spring Batchは初回リリースであるバージョン1が2008年にリリースされました。執筆時点ではバージョン3.0.7が最新となっています。バッチアプリケーションのフレームワークとしては、Java EE 7で標準化されたJSR 352（Batch Applications for the Java Platform、通称jBatch）が知られていますが、この仕様はSpring Batchのアーキテクチャを参考に整備されました。その後、Spring BatchはjBatchのReference Implementationになりました。このようにSpring Batchは、Javaのバッチアプリケーションフレームワークとしての中心的な存在となっています。

COLUMN

Spring Batch と JSR 352 の違い

JSR 352はSpring Batchと非常に似たアーキテクチャとなっていますが、いくつか違いがあります（**表15.1**）。先行して開発されていたSpring Batchのほうが高機能です。Spring BatchはバージョンⅢ系からJSR 352をサポートしており、JSR 352向けのインターフェイスも提供しているため、JSR 352の実装としてSpring Batchを使用することも可能です。

表15.1　Spring Batch と JSR-352 の代表的な相違点

相違点	Spring Batch	JSR 352
パッケージ名	`org.springframework.batch`	`javax.batch`
一部のクラス名	`Tasklet`、`JobExecutionListener`など	`Batchlet`、`JobListener`など
定義ファイル	XMLによるBean定義ファイル、もしくはJava Config	ジョブXMLファイル
`ItemReader`、`ItemWriter`、`ItemProcessor`のジェネリクス対応	あり	なし
`ItemReader`、`ItemWriter`、`ItemProcessor`の標準実装の提供	ファイル入出力、データベース入出力、JMS入出力など	なし

15.1.4 Spring Batchの基本構造

早速、Spring Batchを使用したアプリケーションを作っていきたいのですが、その前にSpring Batchの基本的な構造を把握しておきましょう。Spring Batchはバッチ処理の構造を定義しているため、この構造を理解してか

ら実装を行なうと、自然とバッチ処理のポイントを抑えることができるようになります（図15.1、表15.2）。

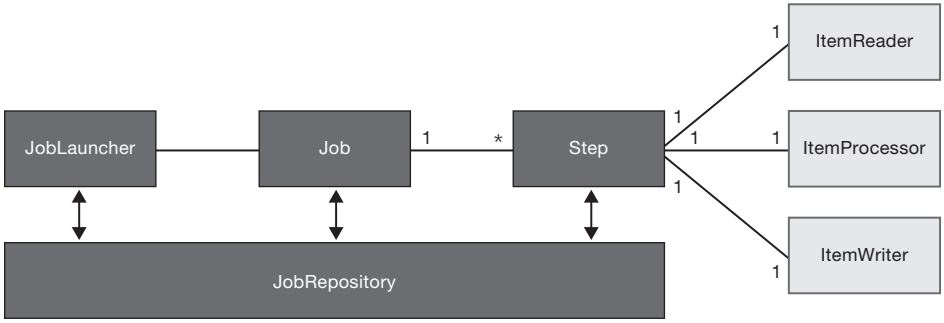

図 15.1 Spring Batch に登場する主な構成要素

表 15.2 Spring Batch に登場する主な構成要素（アプリケーションの実行）

構成要素	役割
JobLauncher	バッチアプリケーションを起動するためのインターフェイス。すべてのバッチアプリケーションはこのクラスから処理が開始する。バッチアプリケーションへ引数を渡すことも可能。JobLauncherを利用者が直接利用することも可能だが、javaコマンドからCommandLineJobRunnerを起動することでより簡単にバッチ処理を開始させることができる。CommandLineJobRunnerは、JobLauncherを起動するための各種処理を引き受けてくれる。
Job	Spring Batchにおけるバッチアプリケーションの一連の処理をまとめた1実行単位。
Step	Jobを構成する処理の単位です。1つのJobに1～N個のStepを持たせることが可能。1つのJobの処理を複数のStepに分割することにより、処理の再利用、並列化、条件分岐といった制御が可能になる。Stepは、チャンク方式またはタスクレット方式のいずれかの方式で実装する。いずれもSpring Batchが定義する方式で、チャンク方式とは、一定件数のデータごとにまとめて入力／加工／出力する方式を指す。タスクレット方式は自由に処理を記述する方式を指す。
ItemReader ItemProcessor ItemWriter	Stepの処理を、データの入力／加工／出力の3つに分割するためのインターフェイス。バッチアプリケーションは、この3パターンの処理で構成されることが多いことに由来し、Spring Batchでは主にチャンク方式でこれらのインターフェイスの実装を活用する。利用者はビジネスロジックをそれぞれの役割に応じて分散して記述する。データの入出力を担うItemReaderとItemWriterは、データベースやファイルからJavaオブジェクトへの変換、もしくはその逆の処理であることが多いため、Spring Batchから標準的な実装が提供されている。ファイルやデータベースからデータの入出力を行なう一般的なバッチアプリケーションの場合は、Spring Batchの標準実装をそのまま使用するだけで要件を満たせるケースもある。
JobRepository	JobやStepの状況を管理する機構。これらの管理情報は、Spring Batchが規定するテーブルスキーマをもとにデータベース上に永続化される。

非同期実行のサポート

JobLauncherのデフォルトの具象クラスであるSimpleJobLauncherは、デフォルト設定を変更することにより、バッチ処理を非同期処理で実行できるようになります。バッチ処理の実行のための実装はTaskExecutorインターフェイスによって抽象化されており、SimpleJobLauncherに設定されているTaskExecutorをデフォルトのSyncTaskExecutorクラスからSimpleAsyncTaskExecutorクラスなどに変更することで実現できます。

15.2 簡単なバッチアプリケーションの作成

ここでは、Spring Batchを用いた簡単なバッチアプリケーションを見ていきます。前節で説明した内容が、どのようにソースコードで表現されるのか確認しましょう。

15.2.1 作成するバッチアプリケーションの要件

CSVファイルに記述されているRoomオブジェクトの情報を1行ずつ読み取り、データベースへインポートするアプリケーションを作成します。CSVファイルは、コマンドライン引数から指定できるようにします。インポート前にデータベース上のテーブルを空にしておく準備処理もバッチ処理の一部として実装していきます。Bean定義の実装にはJava Configを用います。

▶ **本バッチアプリケーションで読み込むCSVファイルのイメージ**

```
roomId, roomName, capacity
1,room A,20
2,room B,10
3,room C,30
...
```

CSVファイルのデータが破損しているなどの理由でデータを正しく読み込めない場合は、本バッチアプリケーションでは異常終了とします。破損データを手動で修復した後にバッチアプリケーションのリスタート（再実行）を行なう運用対処を想定し、リスタート時の開始ポイントはファイルの最初ではなく、中断したデータ行から処理を再開するようにします。

15.2.2 設計

本バッチ処理は、大きく2つの処理に分けられます。

- roomテーブルのレコード全件削除
- CSVファイルからのデータ読み取り、roomテーブルへの書き込み

Spring Batchにおける処理の分割のコツは、処理の主軸となるデータを見つけることです。今回の場合は、CSVファイルの1レコードに相当するRoomオブジェクトが処理の主軸となるデータです。このように、主軸となるデータに対する処理を分割の単位とすると、Spring Batchが定型化している処理の流れに当てはめやすくなります。

一方、主軸となるデータが存在しない処理の場合もあります。たとえば、テーブルのレコード全件削除をTRUNCATEのような単発のコマンドで実行する場合です。主軸となるデータが存在しない処理も、データを意味

のある状態に変化させる部分で分割することができます。

この分割した処理がStepに相当します。今回は2つのStepで1つのJobとし、truncateStep→importFileStepの順序で処理します（**図15.2、表15.3**）。

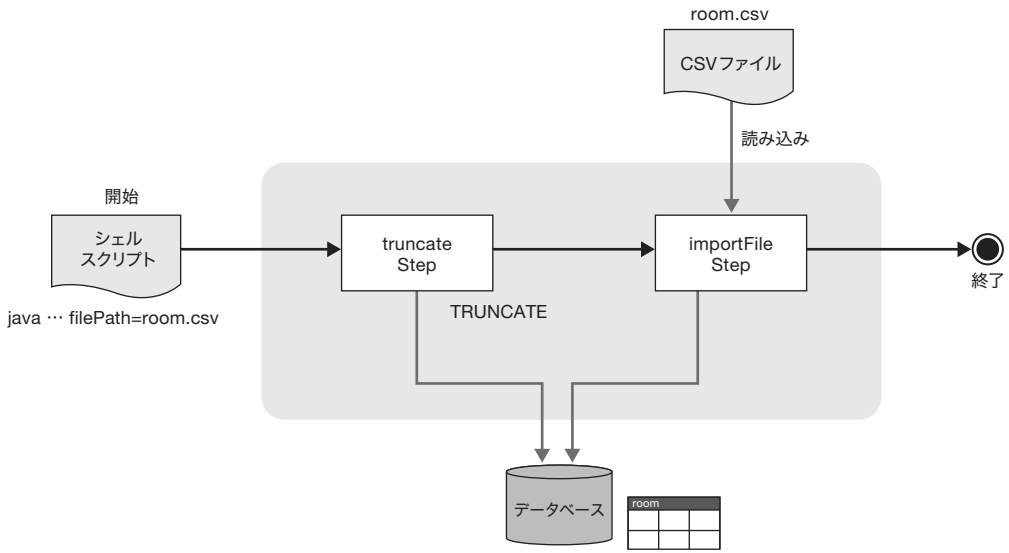

図15.2　処理の全体像

表15.3　処理の分割イメージ

処理順序	Step名	処理内容
1	truncateStep	roomテーブルのレコード全件削除
2	importFileStep	CSVファイルからのデータ読み取り、roomテーブルへの書き込み

15.2.3 pom.xmlの設定

Spring Batchアプリケーションに必要なライブラリの定義と、コマンドラインからアプリケーションを実行する際に使用するExec Maven Pluginの設定を行ないます。

▶ **pom.xmlの定義**

```
<dependencies>
    <dependency>
        <groupId>org.springframework.batch</groupId>
        <artifactId>spring-batch-core</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.batch</groupId>
        <artifactId>spring-batch-test</artifactId>
```

```
        <scope>test</scope>
    </dependency>
    <!-- 中略: その他データベースアクセスに必要なライブラリなど -->
</dependencies>
```

Spring Batchを使用するアプリケーションは、spring-batch-coreを<dependency>に定義します。Spring Batchが提供するテスト機能を使用する場合はspring-batch-testを追加します。その他の、Springアプリケーションに必要な<dependency>についてはここでは割愛します。

次に、コマンドラインからバッチアプリケーションの起動を容易に動作確認できるようにExec Maven Pluginの設定を行います。コマンドラインからSpring Batchアプリケーションを実行する場合は、CommandLineJobRunnerクラスのmainメソッドを呼び出すようにします。コマンドライン引数は<argument>で指定することができます。引数に関する詳細な説明は次節で解説するので、ここでは何を指定しているかについてだけ説明しておきます。第2引数に起動するJobの名前を、第3引数にfilePath引数名でファイルパスを、第4引数にexecutedTimeという引数名で実行時の時刻を指定しています。

▶ **pom.xmlの定義**

```
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>exec-maven-plugin</artifactId>
            <version>1.4.0</version>
            <executions>
                <execution>
                    <goals>
                        <goal>java</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <mainClass>org.springframework.batch.core.launch.support.CommandLineJobRunner</mainClass>
                <arguments>
                    <argument>com.example.app.importfile.ImportFileConfig</argument>
                    <argument>importFileJob</argument>
                    <argument>filePath=file:/tmp/rooms.csv</argument>
                    <argument>executedTime=201511161000</argument>
                </arguments>
                <systemProperties>
                    <systemProperty>
                        <key>spring.profiles.active</key>
                        <value>prod</value>
                    </systemProperty>
                </systemProperties>
            </configuration>
        </plugin>
    </plugins>
</build>
```

15.2.4 Jobの実装

今回作成するJobの名称はimportFileJobとします。Jobの作成は、JobBuilderFactoryを用いてBuilderパターンで作成することができます。これを利用して、Jobを構成するStepを指定します。また、今回のアプリケーションはコマンドラインから引数を受け付けるため、コマンドライン引数をチェックを設定してみましょう。

▶ **Jobの定義**

```
@Configuration
@EnableBatchProcessing ──────────────────────────────────────────────── ❶
@Import({InfrastructureConfig.class, JpaInfrastructureConfig.class})
public class ImportFileConfig {

    public static final String JOB_NAME = "importFileJob";

    @Autowired
    JobBuilderFactory jobBuilderFactory; ──────────────────────────────── ❷

    @Bean
    public JobParametersValidator jobParametersValidator() {
        String[] requiredKeys = new String[]{"filePath"}; ──────┐
        String[] optionalKeys = new String[]{"executedTime"};   ├──── ❸
        return new DefaultJobParametersValidator(requiredKeys, optionalKeys); ┘
    }

    @Bean
    public Job importFileJob() throws Exception {
        return jobBuilderFactory.get(JOB_NAME) ──────────────────────── ❹
                .validator(jobParametersValidator()) ────────────────── ❺
                .start(truncateStep()) ──────────────────────────────── ❻
                .next(importFileStep()).build(); ────────────────────── ❼
    }
}
```

❶ Spring Batchに必要な機能を簡単に利用するために、@EnableBatchProcessingをJava Configクラスに付与する。このアノテーションにより、JobBuilderFactoryなどのSpring Batchに必要なBeanが自動的に生成される

❷ @EnableBatchProcessingによって、生成されたJobBuilderFactoryをインジェクションする

❸ コマンドライン引数をチェックする設定である。必須の引数が指定されていること、もしくは任意の引数以外が入ってこないことをチェックするためには、DefaultJobParametersValidatorを使用する。ここでは、filePathを必須の引数、executedTimeを任意の引数として登録する。Spring Batchは引数の内容によって1回のジョブ実行単位として管理しているため、executedTimeのような実行時ごとに異なる引数を指定することで別のジョブ実行として扱う手法を用いることがある

❹ ジョブの設定を行なう。最初に、ジョブを一意に識別する名称を指定する

❺ ❹に続いて、❸で定義したチェックを使用するように指定する

❻ 1番目のStepにtruncateStepを指定する

❼ 2番目のStepにimportFileStepを指定し、Jobインスタンスを生成する

15.2.5 truncateStepの実装

StepBuilderFactoryを用いてStepを作成します。Step名とStepの処理の実装としてTaskletの具象クラスを指定します。

▶ truncateStepの実装

```
@Configuration
@EnableBatchProcessing
@Import({InfrastructureConfig.class, JpaInfrastructureConfig.class})
public class ImportFileConfig {

    @Autowired
    StepBuilderFactory stepBuilderFactory; ―――――――――――――――――――――――――――――― ❶

    // ・・・

    @Bean
    public Step truncateStep() {
        return stepBuilderFactory.get("truncateStep") ―――――――――――――――――――――――
                .tasklet(truncateTasklet()).build(); ―――――――――――――――――――――――― ❷
    }

    @Bean
    public MethodInvokingTaskletAdapter truncateTasklet() {
        MethodInvokingTaskletAdapter adapter = new MethodInvokingTaskletAdapter(); ―――――――― ❸
        adapter.setTargetObject(truncateService()); ―――――――――――――――――――――――――
        adapter.setTargetMethod("execute"); ―――――――――――――――――――――――――――― ❹
        return adapter;
    }

    @Bean
    public TruncateService truncateService() {
        return new TruncateServiceImpl(); ―――――――――――――――――――――――――――――― ❹
    }
}

public interface TruncateService {
    ExitStatus execute(); ―――――――――――――――――――――――――――――――――――― ❺
}

public class TruncateServiceImpl implements TruncateService {

    @Autowired
    JdbcTemplate jdbcTemplate;

    public ExitStatus execute() { ―――――――――――――――――――――――――――――――― ❺
        jdbcTemplate.execute("TRUNCATE TABLE room");
        return ExitStatus.COMPLETED;
    }
}
```

❶ @EnableBatchProcessingによって、生成されたStepBuilderFactoryをインジェクションする
❷ ステップの設定を行なう。最初に、ステップを一意に識別する名称を指定する。続いて、実行するtaskletの実装を指定し、Stepを生成する
❸ 主軸となるデータが存在しないケースでは、タスクレット方式でStepを作成する。タスクレット方式ではtaskletインターフェイスの具象クラスをBeanとして生成する必要がある。その具象クラスには、POJOなクラスのメソッドを呼び出すことができるMethodInvokingTaskletAdapterを使用するとよい
❹ 呼び出すメソッドはTruncateServiceを実装した、POJOなクラスであるTruncateServiceImplのexecuteメソッドとする
❺ 呼び出される処理を実装する。ここではroomテーブルをTRUNCATEする。MethodInvokingTaskletAdapterで呼び出すメソッドの戻り値に制限はないが、Spring BatchのExitStatusクラスが持つフィールドの値を返却すると、終了コードを制御できる

15.2.6 importFileStepの実装

truncateStepと同様にStepBuilderFactoryを用いてStepを作成します。importFileStepのような主軸となるデータが存在するStepはチャンク方式を適用します。その際は、Taskletの具象クラスにChunkOrientedTaskletを使用してチャンク処理を行ないます。チャンク処理では、データの入出力の実装にItemReaderとItemWriterインターフェイスを使用します。

▶ importFileStepの実装

```
@Configuration
@EnableBatchProcessing
@Import({InfrastructureConfig.class, JpaInfrastructureConfig.class})
public class ImportFileConfig {

    @PersistenceUnit
    EntityManagerFactory entityManagerFactory;

    // ・・・

    @Bean
    public Step importFileStep() {
        return stepBuilderFactory.get("importFileStep").<Room, Room>chunk(100) ―――――――― ❶
                .reader(fileItemReader(null)) ―――――――― ❷
                .writer(dbItemWriter()).build(); ―――――――― ❸
    }

    @Bean
    @StepScope ―――――――― ❹
    @Value("#{jobParameters['filePath']}") ―――――――― ❺
    public FlatFileItemReader<Room> fileItemReader(String filePath) {

        FlatFileItemReader<Room> fileItemReader = new FlatFileItemReader<>();
        ResourceLoader loader = new DefaultResourceLoader();                    ―― ❻
        fileItemReader.setResource(loader.getResource(filePath));
```

```
        DefaultLineMapper<Room> lineMapper = new DefaultLineMapper<>();                    ❼

        DelimitedLineTokenizer lineTokenizer = new DelimitedLineTokenizer();
        lineTokenizer.setNames(new String[]{"roomId", "roomName", "capacity"});            ❽
        lineMapper.setLineTokenizer(lineTokenizer);

        BeanWrapperFieldSetMapper<Room> fieldSetMapper = new BeanWrapperFieldSetMapper<>();
        fieldSetMapper.setTargetType(Room.class);                                          ❾
        lineMapper.setFieldSetMapper(fieldSetMapper);

        fileItemReader.setLineMapper(lineMapper);
        fileItemReader.setLinesToSkip(1);                                                  ❿
        return fileItemReader;
    }

    @Bean
    @StepScope                                                                             ❹
    public ItemWriter<Room> dbItemWriter() {
        JpaItemWriter<Room> jpaItemWriter = new JpaItemWriter<>();                         ⓫
        jpaItemWriter.setEntityManagerFactory(entityManagerFactory);
        return jpaItemWriter;
    }
}
```

❶ ステップの設定を行う。最初に、ステップを一意に識別する名称を指定する。その後、チャンク方式を使うことを示し、100件ごとのトランザクションとする。これで、ChunkOrientedTasklet を指定したのと同じになる。入出力データは、いずれも Room クラスとする

❷ ItemReader の実装として fileItemReader を指定する。 このとき、引数は Null だが、後述のとおり引数から解決する

❸ ItemWriter の実装として dbItemWriter を指定する

❹ ItemReader や ItemWriter は状態を持つことがあるため、複数ステップで使いまわしてしまうと意図しない挙動になってしまう可能性がある。ファイルを読み終わった状態で動作してしまう場合を想像するとよい。これを避けるために @StepScope を指定すると1ステップを有効なスコープとした Bean が生成される

❺ メソッド引数の String filePath をジョブの引数から解決する設定である。ジョブの引数で filePath=xxx と指定した場合、xxx が値として入る

❻ CSV ファイルからデータ取得を行うため、ItemReader の実装として一般的なテキストファイルを入力とする FlatFileItemReader を指定する。引数で指定したパスにあるファイルを対象とする

❼ ファイルの1行をオブジェクトにマッピングする。DefaultLineMapper は、後述の LineTokenizer と FieldSetMapper の2つが必要になる

❽ LineTokenizer はファイルの1行のデータをどのように解析するかを定義する。DelimitedLineTokenizer はカンマ区切りで分割し、1つ目から順番に "roomId"、"roomName"、"capacity" という名称のフィールドへ格納する

❾ ファイルの1行のデータを解析した結果を、どのようにオブジェクトにマッピングするかを定義する。ここでは、Room クラスにマッピングする

❿ ItemReader にマッピング方法を指示する。同時に、CSV ファイルのヘッダー行をスキップし、残りをデータとして扱うため setLinesToSkip(1) とする

⓫ データベースのテーブルへデータを出力するために JpaItemWriter を使用する。渡されてくる加工済みの Room クラスをそのままテーブルに反映する

15.2.7 テスト

Spring Batch アプリケーションを JUnit テストケースから実行する場合は、JobLauncherTestUtils.launchJob() を使用すると便利です。コマンドライン引数の代わりに、JobParameters をその場で作ります。

▶ テストの実装

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {ImportFileConfig.class})                    ❶
@ActiveProfiles("dev")                                                       ❶
public class JobTest {
    @Autowired
    Job job;                                                                 ❷

    @Autowired
    JobLauncher launcher;                                                    ❷

    @Autowired
    JobRepository jobRepository;                                             ❷

    @Bean                                                                    ❸
    public JobLauncherTestUtils jobLauncherTestUtils() {                     ❸
        JobLauncherTestUtils utils = new JobLauncherTestUtils();             ❸
        utils.setJob(job);                                                   ❸
        utils.setJobLauncher(launcher);                                      ❸
        utils.setJobRepository(jobRepository);                               ❸
        return utils;                                                        ❸
    }                                                                        ❸

    @Test
    public void testJob() throws Exception {
        Map<String, JobParameter> map = new HashMap<>();                     ❹
        map.put("filePath", new JobParameter("rooms.csv"));                  ❹
        map.put("executedTime", new JobParameter("201511161000"));           ❹
        JobParameters params = new JobParameters(map);                       ❹

        BatchStatus status = jobLauncherTestUtils().launchJob(params).getStatus();  ❺
        assertThat(status.name(), is("COMPLETED"));                          ❺
    }
}
```

❶ テスト対象の Java Config クラスと、テスト用 Profiles を指定する

❷ テストに必要となるインスタンスをインジェクションする。今回は Job が 1 つしか定義していないが、複数定義している場合は識別する必要がある

❸ テスト用のジョブ起動クラスである、JobLauncherTestUtils を生成する

❹ コマンドライン引数を格納するクラスである、JobParametersをその場で作り上げる
❺ Jobを起動し、結果が正常終了したことを確認する

先に述べたとおり、Spring BatchはJobの実行を管理するためのテーブルを必要とします。テスト実行時などにはデータベースにH2などの組み込みデータベースを使用する場合、以下のように、組み込みデータベース構築時にSpring Batchが使用するテーブルを作成するようにします。テーブルの定義はSpring Batchのjarファイル内に、各種データベース製品ごとにSQLスクリプトが用意されています。Spring Batch以外のテーブルとしてroomテーブルの作成が必要になります。今回のアプリケーションでは、簡単のためroomテーブルの作成はJPA実装に委ねることとします。

▶ テスト用データソースの定義

```
@Configuration
public class InfrastructureConfig {

    // ・・・

    @Bean
    @Profile("dev")
    public DataSource jobRepositoryEmbeddedDataSource() {
        return new EmbeddedDatabaseBuilder()
                .setType(EmbeddedDatabaseType.H2)
                .addScript(
                        "classpath:org/springframework/batch/core/schema-h2.sql")
                .build();
    }
}
```

なお、本アプリケーションではJPAを用いてデータアクセスを行うため、SpringでJPAを利用するために必要なBeanを定義しておく必要がありますが、本章では割愛します。詳細については、第10章「Spring Data JPA」を参照してください。

それでは、実際にテストケースからジョブを実行してみましょう。正しく実装ができていればJUnitのテストケース1件が成功することが確認できるはずです。

15.2.8 正常系の実行

テストが成功することが確認できたら、実際にコマンドラインからバッチアプリケーションを実行してみましょう。バッチアプリケーションを実行する前に、テスト時の組み込みデータベースと同様に、本番実行用のデータベースにSpring Batchが使用するテーブルを定義します。組み込みデータベースのように、起動ごとにテーブルの内容が初期化されてしまっては困るため、手動でテーブル作成を行います。spring-batch-coreのjarファイルのorg.springframework.batch.coreパッケージ内にあるSQLスクリプトをSQL実行クライアント

ツールなどを用いて実行します。たとえばPostgreSQLを使用する場合は、schema-postgresql.sqlファイルをjarから抽出し、psqlなどで実行します。

テーブルの作成が終わったら、mvn packageコマンドで本アプリケーションをビルドし、コマンドラインから本バッチアプリケーションを実行してみます。実行後にechoコマンドでJVMプロセスの終了コードが0（正常終了）であることを確認しています。

▶ バッチアプリケーションのビルドおよび実行

```
$ mvn clean package
$ mvn exec:java
$ echo $?
0
```

roomテーブルにCSVファイルのデータが書き込まれていることが確認できます（**表15.4**）。

表 15.4　バッチアプリケーション実行後の room テーブル

room_id [PK]	capacity	room_name
1	20	room A
2	10	room B
3	30	room C
4	21	room A
5	11	room B
…	…	…

Spring BatchがJobの実行を管理するためのメタデータも、Spring Batchの各テーブルから確認することができます（**表15.5**）。batch_step_executionテーブルからはStepの実行結果を確認することができますが、いずれのStepも正常終了（COMPLETED）していることがわかります。

表 15.5　バッチアプリケーション実行後の batch_step_execution テーブル

step_execution_id [PK]	step_name	job_execution_id [FK]	status	commit_count	read_count	write_count	rollback_count	exit_code	exit_message
1	truncateStep	1	COMPLETED	1	0	0	0	COMPLETED	
2	importFileStep	1	COMPLETED	6	500	500	0	COMPLETED	

なお、Spring Batchでは正常終了したバッチ処理をリスタートすることはできません。Spring Batchでは同一のコマンドライン引数が指定されたバッチ処理は前回実行したジョブのリスタートとして認識されるため、まったく同じ引数で実行しようとするとJobInstanceAlreadyCompleteExceptionが発生します。再度同じCSVファイルの先頭から処理をしなおしたい場合は、executedTimeの値を前回実行時と違う値に変更するなどの対応が必要になります。

15.2.9 異常系の実行

次に、エラーデータが含まれるCSVを読み込んだ際に、バッチ処理が異常終了し、エラーデータの修復後にエラーデータから処理がリスタートできるかを確認します。

CSVファイルの任意のデータを意図的に破壊します。 ここでは以下のように、128番目のデータのカラムを削ります。

▶ エラーを含んだCSVファイルの例

```
roomId, roomName, capacity
...
127,room A,62
128,room B
129,room C,72
...
```

リスタートを回避するためにexecutedTimeをインクリメントするのを忘れずにバッチアプリケーションを実行すると、エラーデータの存在によりFlatFileParseExceptionが発生し、JVMプロセスが異常終了して終了コード1が返却されます。

▶ バッチアプリケーションの実行

```
$ mvn exec:java
$ echo $?
1
```

この時点でのroomテーブルの状態を確認すると、エラーデータが含まれる前のチャンクのデータ（100行目）までがテーブルに出力されています（**表15.6**）。読み込まれていたはずのエラーデータが含まれるチャンクのデータ（101～127件目）のデータはデータベースへ書き込みが行われていないことがわかります。

表15.6　バッチアプリケーション異常終了直後のroomテーブル

room_id [PK]	capacity	room_name
…	…	…
93	60	room C
94	51	room A
95	41	room B
96	61	room C
97	52	room A
98	42	room B
99	62	room C
100	53	room A

batch_step_execution テーブルからは、truncateStep は正常終了したが importFileStep は異常終了（FAILED）していることがわかります（**表 15.7**）。他にも何件目のデータまで読み込みを行っていて、何件目のデータまで書き込みが完了しているかが記録されていることがわかります。

表 15.7　バッチアプリケーション異常終了直後の batch_step_execution テーブル

step_execution_id [PK]	step_name	job_execution_id [FK]	status	commit_count	read_count	write_count	rollback_count	exit_code	exit_message
1	truncateStep	1	COMPLETED	1	0	0	0	COMPLETED	
2	importFileStep	1	COMPLETED	6	500	500	0	COMPLETED	
3	truncateStep	2	COMPLETED	1	0	0	0	COMPLETED	
4	importFileStep	2	FAILED	1	127	100	1	FAILED	FlatFileParseException ...（スタックトレース略）

次に、CSV のエラーデータを修復し、バッチアプリケーションをリスタートしてみます。リスタートを行う場合は、必ず前回実行時と同一のコマンドライン引数の値を指定する必要があります。 そのため、executedTime をインクリメントせずにバッチアプリケーションを実行します。JVM プロセスが正常終了していることが確認できます。

▶ **バッチアプリケーションの実行**

```
$ mvn exec:java
$ echo $?
0
```

room テーブルにも全件データが書き込まれていることが確認できます（**表 15.8**）。batch_step_execution テーブルを確認すると、前回正常終了していた truncateStep は実行されず、前回異常終了した importFileStep のみが新たに実行されていることがわかります。読み込んだデータと書き込んだデータの件数は、エラーデータの出現により書き込みが完了しなかったデータ含めた未処理のデータの件数と一致しています。

表 15.8　エラーデータ修復後のリスタート直後の batch_step_execution テーブル

step_execution_id [PK]	step_name	job_execution_id [FK]	status	commit_count	read_count	write_count	rollback_count	exit_code	exit_message
1	truncateStep	1	COMPLETED	1	0	0	0	COMPLETED	
2	importFileStep	1	COMPLETED	6	500	500	0	COMPLETED	
3	truncateStep	2	COMPLETED	1	0	0	0	COMPLETED	
4	importFileStep	2	FAILED	1	127	100	1	FAILED	FlatFileParseException ...（スタックトレース略）
5	importFileStep	3	COMPLETED	5	400	400	0	COMPLETED	

このように、ビジネスロジックでリスタートを意識しなくても、Spring Batch の枠組みに従って実装を行うことで Step の単位やチャンクの単位でエラー後のリスタートを実現することができます。

15.3 Spring Batchのアーキテクチャ

前節ではシンプルなバッチアプリケーションをSpring Batchを用いて実装することで、どのように活用するのかを理解できたと思います。本節ではSpring Batchが持つさまざまな機能を順に見ていきたいと思います。

15.3.1 構成要素の詳細

前節までにSpring Batchの基本構造については簡単に説明しましたが、その他にも重要な要素があります。ここでは、前節までに説明した全体の処理の流れに加え、ジョブの実行状況などのメタデータがどのように管理されているかについてもより詳細に把握していきましょう。**図15.3**をもとに説明していきます。

バッチ処理をエンタープライズで構築する場合、多数のジョブ同士に先行／後続関係を定義して1つのユースケースを実現するのが一般的です。このとき、ジョブスケジューラを使用してバッチ処理の実行管理を行なうことがよくあります。

一般に、ジョブスケジューラは以下の機能を搭載しています。

- **ジョブフロー**
  先行ジョブの処理結果に応じて後続ジョブを起動し分けたり、複数のジョブを並列実行したりといった柔軟な処理の流れを定義できる
- **スケジューリング**
  ジョブの開始時刻や実行サイクルを定義できる
- **実行管理**
  正常終了したジョブの履歴を確認したり、異常終了したジョブを再実行したりすることができる

ジョブスケジューラからジョブを起動する場合、その都度Javaプロセスを新たに起動し、処理が終わり次第プロセスを終了させる実行方法が普通です。

ジョブをSpring BatchのStepと比較すると、Spring Batchがジョブスケジューラが提供する前述した役割に近い機能を提供しています。たとえばSpring Batchでは、Stepのフロー制御を行ったり、Stepの単位での実行管理を行なったりすることができます。言い換えると、Spring Batchはジョブスケジューラの役割の一部を担うことができるということです。これらの機能がフレームワークで実現されることは大変喜ばしいことではありますが、一方でSpring Batchはジョブの状態や結果などのメタデータをバッチ処理の内外で正しく管理している必要があります。以下では、Spring Batchがそれらの情報をどのように管理しているかを確認していきます（**表15.9**）。

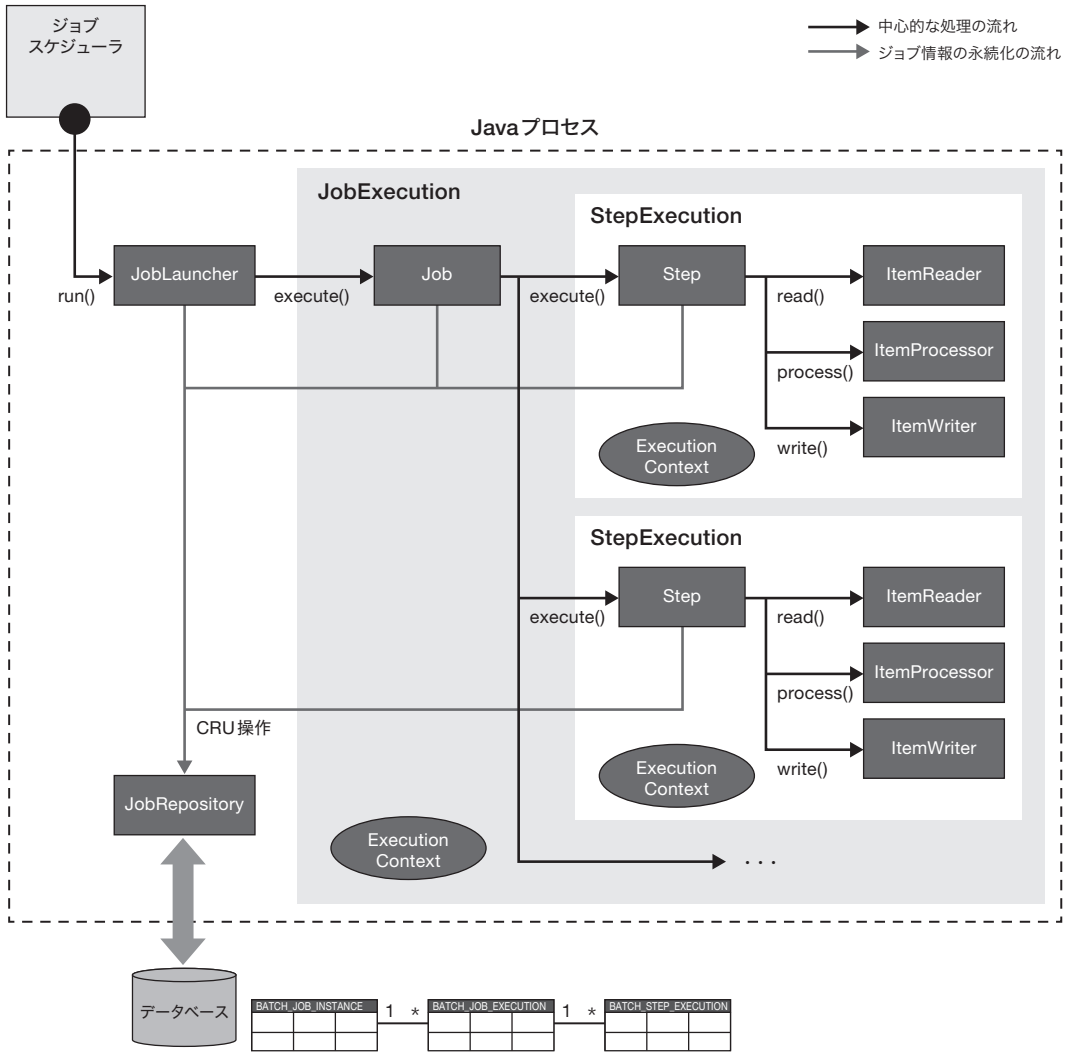

図 15.3 Spring Batch の主な構成要素と処理の流れ

表 15.9　Spring Batch に登場する主な構成要素（アプリケーション内のメタデータ管理）

構成要素	役割
JobInstance	Jobの「論理的」な実行を示す。最後までJobを実行した後に、再度Jobを実行した場合は別のJobInstanceとなる。一方、Jobを実行し、処理の途中でエラーなので中断をしたあと再度同一のJobを途中から実行しなおすような場合は、初回の実行と同一のJobInstanceとみなされる。
JobExecution ExecutionContext	JobExecutionはJobの「物理的」な実行を示す。JobInstanceとは異なり、同一のJobを再実行する場合も別のJobExecutionとなる。結果、JobInstanceとJobExecutionは1対多の関係になる。同一のJobExecution内で処理の進捗状況などのメタデータを共有したい場合のための領域として、ExecutionContextがある。ExecutionContextは主にSpring Batchがフレームワークの状態などを記録するために使用されているが、アプリケーションがExecutionContextへアクセスする手段も提供されている。
StepExecution ExecutionContext	StepExectuionはStepの「物理的」な実行を示す。JobExecutionとStepExecutionは1対多の関係になる。JobExectuionと同様に、Step内でデータを共有するための領域ExecutionCotnextがある。データの局所化の観点から、複数のJobExecutionで共有する必要のない情報はJobのExecutionContextを使用するのでなく、対象のStep内のExecutionContextを利用したほうがよい。
JobRepository	JobExecutionやStepExecutionなどのバッチアプリケーションの実行結果や状態を管理するためのデータを管理、永続化する機能を提供する。一般的なバッチアプリケーションはJavaプロセスを起動することで処理が開始し、処理の終了とともにJavaプロセスも終了させるケースが多い。そのためこれらのデータはJavaプロセスを跨いで参照される可能性があることから、揮発性なメモリ上だけではなくデータベースなどの永続層へ格納する。データベースに格納する場合は、JobExecutionやStepExecutionを格納するためのテーブルやシーケンスなどのデータベースオブジェクトが必要になる。Spring Batchが提供するスキーマ情報をもとにデータベースオブジェクトを生成する必要がある。

Spring Batchがなぜメタデータをこれほどまでに重厚に管理を行なっているかというと、次の節で紹介するSpring Batchがサポートする再実行性を実現するためです。バッチ処理を再実行可能にするには、前回実行時のスナップショットを残しておく必要があり、メタデータやJobRepositoryはそのための基盤となっています。

15.3.2 Jobの起動

まずは、バッチアプリケーションの起点となる、Jobの起動について確認しましょう。Javaプロセス起動直後にバッチ処理を開始し、バッチ処理が完了後にJavaプロセスを終了するケースを考えます。Javaプロセス起動と同時に、Spring Batch上で定義されたJobを開始するためには、Javaを起動するシェルスクリプトを記述するのが一般的でしょう（**図15.4**）。

Spring Batchが提供するCommandLineJobRunnerを使用すると、利用者が定義したSpring Batch上のJobを簡単に起動することができます。

▶ Java ConfigによるBean定義を行った場合のバッチアプリケーション起動コマンド

```
$ java -cp ${CLASSPATH} org.springframework.batch.core.launch.support.CommandLineJobRunner ➥
<ConfigクラスFQCN> <Job名> <Job引数名1>=<値1> <Job引数名2>=<値2> ...
```

▶ XMLによるBean定義を行った場合のバッチアプリケーション起動コマンド

```
$ java -cp ${CLASSPATH} org.springframework.batch.core.launch.support.CommandLineJobRunner ➥
<XMLファイルのパス> <Job名> <Job引数名1>=<値1> <Job引数名2>=<値2> ...
```

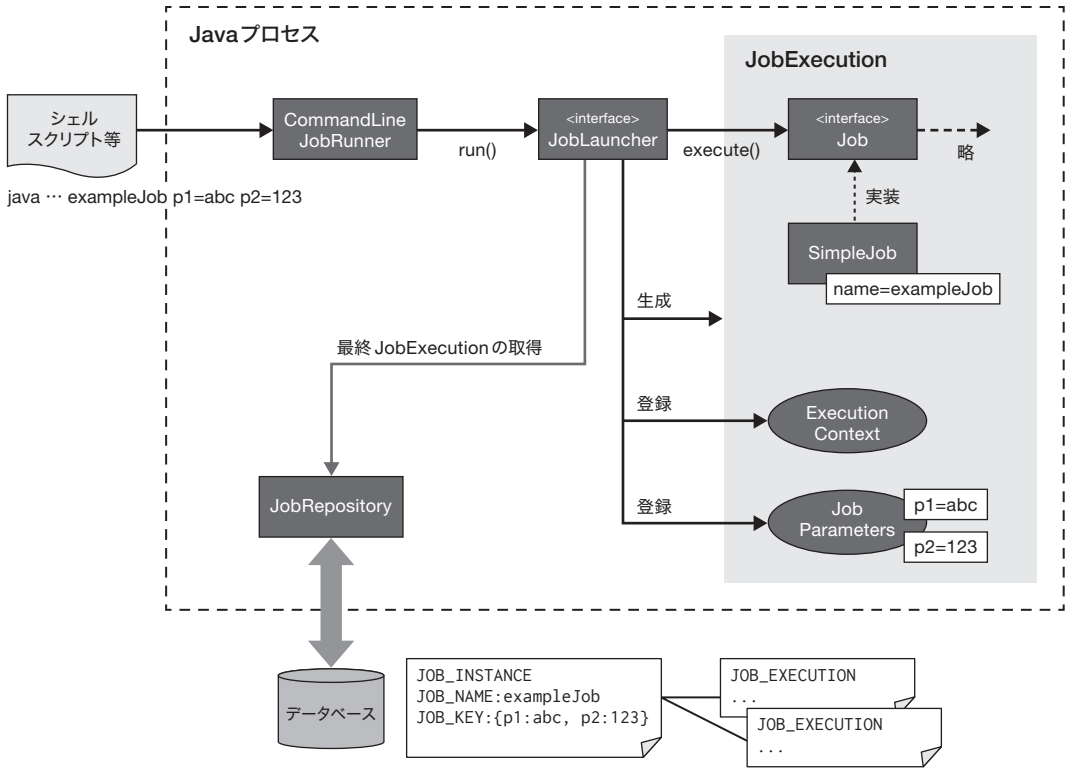

図 15.4　シェルスクリプトからの Job の起動

起動したい Job を指定するために Job 名を指定していますが、Job 名には DI コンテナ上に作成された Job インターフェイスの具象クラスの Bean 名を指定します。Job の具象クラスには、特殊な拡張要件がない限り SimpleJob を使用します。複数の Job を定義した場合は、Bean 名で Job を識別します。

CommandLineJobRunner は起動する Job 名だけでなく、引数を渡すことも可能です。引数は前述した例のように、<Job引数名> = <値> の形式で指定します。すべての引数は CommandLineJobRunner や JobLauncher が解釈とチェックを行なったうえで、JobExecution へ JobParameters に変換して格納します。コマンドラインから引数を介して Java アプリケーションへデータを引き継ぐ方法はバッチアプリケーションでの常套手段なので、活用する機会は多いでしょう。

引数の型指定

Job の引数はデフォルトでは文字列型として認識されますが、引数名の末尾に引数の型を示すことで、引数の型を文字列型以外の型に自動的に変換して扱うことが可能です。以下の型を指定することができます。

- (string)
- (date)

- (long)
- (double)

使用例： operation.date(date)=2015/12/31

Spring Batchでは Jobの引数にさらに特別な意味を持たせています。前節では、Spring Batchがバッチ処理の実行管理を担っており、JobInstanceや JobExecutionなどのバッチ処理のメタデータを永続化して管理していることを説明しました。Spring Batchは Jobの論理的な実行を指す JobInstanceを Job名と引数によって識別しています。言い換えると、Job名と引数が同一である実行は、同一の JobInstanceの実行と認識し、前回起動時の続きとして Jobを実行します。対象の Jobが再実行をサポートしていて、前回実行時にエラーなどで処理が途中で中断していた場合は処理の途中から実行されます。

一方、再実行をサポートしていない Jobや、対象の JobInstanceがすでに正常に処理が完了している場合は例外が発生し、Javaプロセスが異常終了します。たとえば、すでに正常に処理が完了している場合は JobInstanceAlreadyCompleteExceptionが発生します。**図15.9**に示しているように、JobLauncherが JobRepositoryから Job名と引数に合致する JobInstanceをデータベースから取得し、該当する JobInstanceが存在した場合は、紐付いている JobExecutionを復元します。

そのため、Spring Batchでは日次実行など繰り返して起動する可能性のある Jobに対しては、JobInstanceがユニークにするためだけの引数を追加する方法がとられています。たとえば、システム時刻であったり、乱数を引数に追加する方法が挙げられます。

COLUMN

同じジョブを何度も手軽に実行するには

引数を一致させて JobInstanceを同一視させて再実行を行なうのは面倒に感じるかもしれません。この仕様を回避する方法として、CommandLineJobRunnerの -nextオプションを使用する方法があります。-nextオプションを使用すると、CommandLineJobRunner内部で引数を追加し、機械的に生成した連番を割り当てるためコマンドラインから渡す引数が同一であっても別の JobInstanceとして認識されます。ただし、-nextオプションの使用を前提にしてしまうと JobInstanceを特定しにくくなり、復旧に時間を要す可能性が高まるため注意が必要です。 このオプションによる対処の他には、引数に実行日時を渡す、または CommandLineJobRunnerが内部的に利用している JobParametersConverterの具象クラスに JsrJobParametersConverterを使うといった方法があります。

15.3.3 ビジネスロジックの定義

Spring Batchでは、ビジネスロジックをStepと呼ばれるさらに細かい単位に分割することができます。Jobが起動すると、Jobは自身に登録されているStepを起動し、StepExectuionを生成します。Stepはあくまで処理の分割のための枠組みのみを提供しており、ビジネスロジックの実行はStepから呼び出されるTaskletに任されています。**図15.5**にStepからTaskletへの処理の流れを示します。

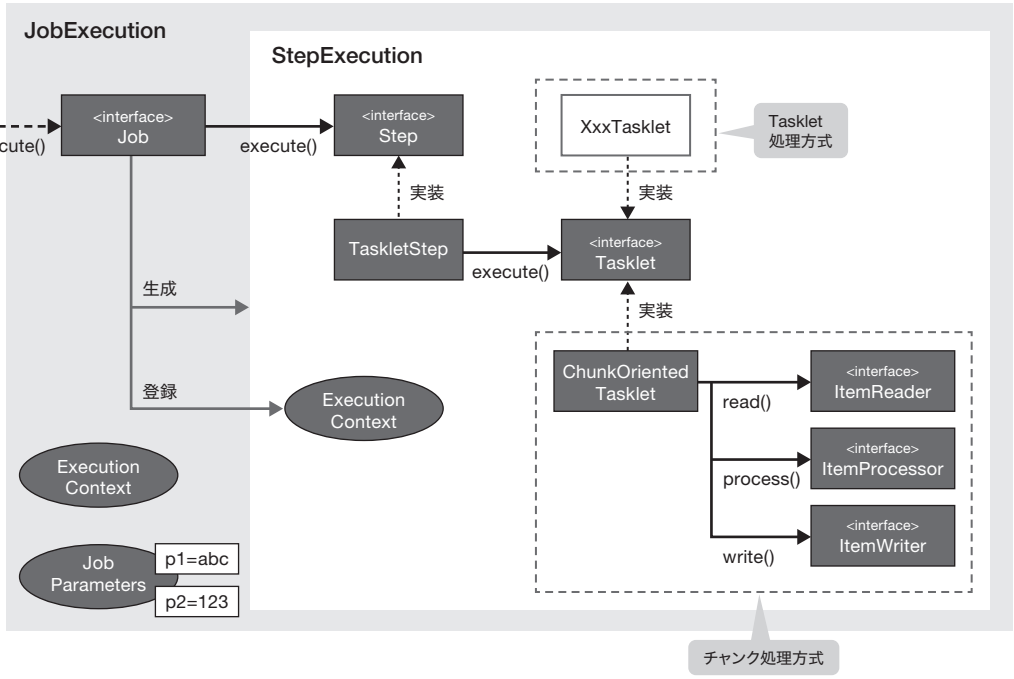

図15.5 StepとTasklet

このように、Taskletの利用方法には2つの方式があります。「チャンク方式」と「タスクレット方式」です。概要についてはすでに説明しているので、ここではその構造について見ていきましょう。

■チャンク方式

前述したようにチャンク方式とは、処理対象となるデータを1件ずつ処理するのではなく、一定数の塊（チャンク）の単位で処理を行なう方式です。ChunkOrientedTaskletがチャンク処理をサポートしたTaskletの具象クラスとなっており、commit-intervalという設定値により、チャンクに含めるデータの最大件数を調整することができます。ItemReader、ItemProcessor、ItemWriterは、いずれもチャンク処理を前提としたインターフェイスとなっています。

次に、ChunkOrientedTaskletがどのようにItemReader、ItemProcessor、ItemWriterを呼び出しているかを確

認し、チャンク処理への理解をより深めていきましょう。以下に、ChunkOrientedTasklet が1つのチャンクの処理を行なう際のシーケンス図を示します（**図15.6**）。

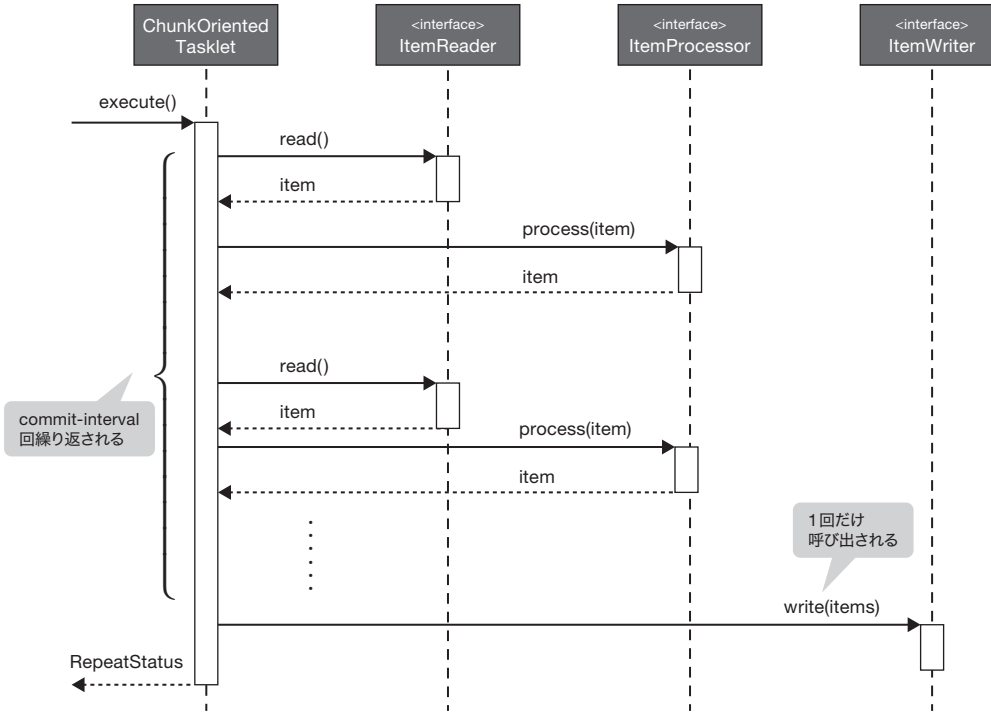

図15.6　1つのチャンクに対する処理

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

チャンクの中のデータの一部にエラーが含まれている場合に、バッチ処理全体を中断するのではなく、エラーデータに対する処理だけをスキップしたいことがバッチ処理ではよくあります。このようなことも Spring Batch では簡単に実現できます。発生した例外の型によってスキップするか、Step 自体を中断するかを判断させることができます。チャンク処理を行なう Step の定義例を以下に示します。

▶ **XMLによるBean定義例**

```
<batch:job id="job">
    <batch:step id="step1">
        <batch:tasklet>
```

```
            <!-- ItemReader, ItemProcessor, ItemWriterのBeanを指定 -->
            <batch:chunk reader="myItemReader"
                processor="myItemProcessor" writer="myItemWriter"
                commit-interval="100">
                <batch:skippable-exception-classes>
                    <!-- 例外発生時にStepを異常終了せず処理をスキップさせたい例外 -->
                    <batch:include class="java.lang.IllegalStateException" />
                    <batch:include class="java.lang.NumberFormatException" />
                </batch:skippable-exception-classes>
            </batch:chunk>
        </batch:tasklet>
    </batch:step>
</batch:job>
```


■各インターフェイスで提供されている実装

ここでは、チャンク処理において実際の処理を担うItemReader、ItemProcessor、ItemWriterについて紹介していきます。各インターフェイスとも利用者が独自に実装を行なうことが想定されていますが、Spring Batchが提供する汎用的な具象クラスでまかなうことができる場合があります。次に、Spring Batchから提供されている具象クラスを示します（**表15.10**）。

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

▶ **ItemProcessorの実装例**

```
public class MyItemProcessor implements
        ItemProcessor<MyInputObject, MyOutputObject> {
    @Override
    public MyOutputObject process(MyInputObject item) throws Exception {
        // 入力データであるitemに対してビジネスロジックを記述します。

        return processedObject;  // ビジネスロジックで処理した結果を返却します。
    }
}
```

ItemReaderやItemWriterはさまざまな具象クラスがSpring Batchから提供されていますが、特殊な形式のファイルを入出力したりする場合は、独自のItemReaderやItemWriterを実装した具象クラスを作成し使用することができます。その際、ファイルアクセスのように、入出力前後にopenやcloseなどの初期化処理、解放処理が必要な場合は、ItemStreamインターフェイスを合わせて実装するか、ItemStreamReaderかItemStreamWriterを実装します。ItemStreamインターフェイスは初期化や解放処理を実装するためのメソッドであるopen、closeメソッドが用意されており、Spring BatchからStepの処理開始時、終了時に呼び出されます。

表 15.10　Spring Batch が提供する ItemReader、ItemProcessor、ItemWriter の代表的な具象クラス

インターフェイス	具象クラス名	概要
ItemReader	FlatFileItemReader	CSV ファイルなどの、非構造的なファイルの読み込みを行なう。Resource オブジェクトをインプットとし、区切り文字やオブジェクトへのマッピングルールをカスタマイズすることができる
	StaxEventItemReader	XML ファイルの読み込みを行なう。名前の通り、StAX をベースとした XML ファイルの読み込みを行う実装となっている
	JdbcPagingItemReader JdbcCursorItemReader	JDBC を使用して SQL を実行し、データベース上のレコードの読み込みを行なう。データベース上の大量のデータを処理する場合は、全件をメモリ上に読み込むことを避け、一度の処理に必要なデータのみの読み込み、破棄を繰り返す必要がある。JdbcPagingItemReader は JdbcTemplate を用いて SELECT SQL をページごとに分けて発行することで実現する。一方、JdbcCursorItemReader は JDBC のカーソルを使用することで、1 回の SELECT SQL の発行で実現する
	JpaPagingItemReader HibernatePagingItemReader HibernateCursorItemReader	JPA 実装や Hibernate などと連携してデータベース上のレコードの読み込みを行います。iBAITS（現 MyBatis）と連係する具象クラスもあったが、現在は他の iBATIS 連携機能と同様に非推奨クラスとなっている。その代わりに、MyBatis が提供している Spring 連携ライブラリ MyBatis-Spring から、org.mybatis.spring.batch.MyBatisPagingItemReader が提供されている
	JmsItemReader AmqpItemReader	JMS や AMQP からメッセージを受信し、その中に含まれるデータの読み込みを行なう
ItemProcessor	PassThroughItemProcessor	何も行なわない。入力データの加工や修正が不要な場合に使用する
	ValidatingItemProcessor	入力チェックを行なう。入力チェックルールの実装には、Spring Batch 独自の org.springframework.batch.item.validator.Validator を実装する必要があるが、Spring から提供されている汎用的な org.springframework.validation.Validator へのアダプタである SpringValidator が提供されており、org.springframework.validation.Validator のルールを流用できる
	CompositeItemProcessor	同一の入力データに対し、複数の ItemProcessor を逐次的に実行する。ValidatingItemProcessor による入力チェックの後にビジネスロジックを実行したい場合などに有効
ItemWriter	FlatFileItemWriter	処理済みの Java オブジェクトを、CSV ファイルなどの非構造的なファイルとして書き込みを行なう。区切り文字やオブジェクトからファイル行へのマッピングルールをカスタマイズできる
	StaxEventItemWriter	処理済みの Java オブジェクトを XML ファイルとして書き込みを行なう
	JdbcBatchItemWriter	JDBC を使用して SQL を実行し、処理済みの Java オブジェクトをデータベースへ出力する。内部では JdbcTemplate が使用されている
	JpaItemWriter HibernateItemWriter IbatisBatchItemWriter	JPA 実装や Hibernate などと連携して、処理済みの Java オブジェクトをデータベースへ出力する
	JmsItemWriter AmqpItemWriter	処理済みの Java オブジェクトを、JMS や AMQP でメッセージを送信する

また、ItemStream には StepExecution の ExecutionContext へアクセスできるメソッドが用意されています。そのため、前回のバッチ処理実行時に処理を行ったデータの最終位置を ExecutionContext から取得し、そのデータまでスキップすることでリスタート可能なバッチ処理を実現することが可能になります。ExecutionContext は、open、close に加え、1つのチャンクの処理が完了するたびに呼び出される update メソッドの中でアクセスすることができます。

▶ ItemStreamReaderの実装例

```
public class MyItemStreamReader implements ItemStreamReader<MyInputObject> {

    @Override
    public void open(ExecutionContext executionContext)
            throws ItemStreamException {
        // Step処理開始時に呼び出されます。リソースの初期化処理などを行います。
    }

    @Override
    public void update(ExecutionContext executionContext)
            throws ItemStreamException {
        // 1つのチャンク処理が終わるたびに呼び出されます。executionContextの更新などを行います。
    }

    @Override
    public void close() throws ItemStreamException {
        // Step処理終了時に呼び出されます。リソースの解放処理などを行います。
    }

    @Override
    public MyInputObject read() throws Exception, UnexpectedInputException, ParseException,
            NonTransientResourceException {
        // データー件を読み込む処理を行います。
        return readObject;  // 読み込んだデータを出力します。
    }
}
```

■タスクレット方式

前述のチャンク処理は、複数の入力データを1件ずつ読み込み、一連の処理を行うバッチアプリケーションに適した枠組みとなっていました。しかし、時にはチャンク処理の型に当てはまらないような処理を実装することもあります。たとえば、システムコマンドを実行したり、制御用のテーブルのレコードを1件だけ更新したいような場合などです。そのような場合には、チャンク処理によって得られる性能面のメリットが少なく、設計の難しさによるデメリットの方が大きいため、タスクレット方式を使用するほうが合理的です。

タスクレット方式を使用する場合は、システムコマンドを実行するための SystemCommandTasklet や POJO で記述されたクラスの特定のメソッドを実行する MethodInvokingTaskletAdapter などの Spring Batch が提供する Tasklet の具象クラスを利用します。Spring Batch から提供されている Tasklet インターフェイスを利用者が実

装する方法がありますが、業務ロジックのSpring Batchへの依存度を下げたい場合は前者の具象クラスを使用したほうがよいでしょう（**表15.11**）。

表 15.11　Spring Batch が提供する Tasklet の代表的な具象クラス

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

COLUMN

Spring Batch 独自の Bean スコープ

Spring Batchには独自のBeanスコープが用意されています。stepスコープとjobスコープです。 それぞれStepやJobの中でライフサイクルが完結するシングルトンのBeanです。たとえば、ItemReader、ItemProcessor、ItemWriterなどは、読み書きしているファイルなどのリソース情報の状態を保持するステートフルなBeanであるため、同一のBeanを複数のStepで使いまわすことは避けるべきです。stepスコープでBean定義をすると、異なるStepでは新たなBeanとして別のインスタンスが生成されるため、安全にステートフルなBeanを扱うことができます。Java Config向けに@StepScopeや@JobScopeが用意されています。

また、これらのスコープを使用すると、JobやStepの属性をレイトバインディングすることが可能になります。レイトバインディングを使用すると、実行時に値が決まる属性の値を、SpELを用いてBeanのプロパティに設定できます。たとえばJobParameterに含まれるファイルパスを、FlatFileItemReaderのBean定義のプロパティに指定できます。

▶ **stepスコープのBeanに対するレイトバインディングの使用例**

```
<bean id="flatFileItemReader" scope="step"
      class="org.springframework.batch.item.file.FlatFileItemReader">
    <property name="resource" value="#{jobParameters['input.file.name']}" />
</bean>
```

レイトバインディングの詳細については、以下のURLを参照してください。

http://docs.spring.io/spring-batch/trunk/reference/html/configureStep.html#late-binding

15.3.4 トランザクション管理

Springでトランザクションを管理するための機能として、PlatformTransactionManagerやTransactionTemplateがありますが、Spring Batchでもそれらの仕組みを使用してトランザクション管理を行ないます（**図15.7**）。ただし、Spring Batch自身にトランザクション管理が組み込まれているため、オンラインアプリケーションのように実装するアプリケーション側でトランザクションを宣言したり操作を行なったりする必要はありません。特にチャンク処理の場合は、1つのチャンクに対する一連の処理が1つのトランザクションとなるため、チャンク処理の枠組みを提供するSpring Batchがトランザクション管理の責務を負う必要があります。

Spring Batchはデフォルトで、TaskletやItemReader、ItemProcessor、ItemWriterで任意の例外が発生した場合にロールバックを行います。当然ながら、発生した例外がスキップ対象に設定されている例外の場合は、ロールバックの対象となりません。

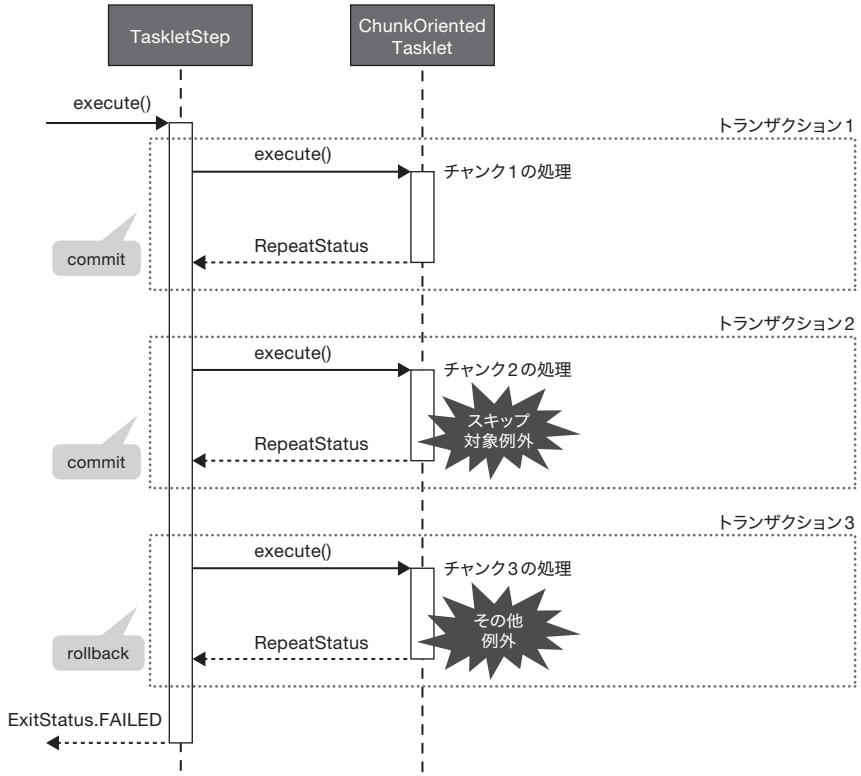

図15.7　チャンク処理のトランザクション範囲

15.3.5 バッチ処理結果の出力

バッチ処理では、アプリケーションの処理結果を画面に表示することができないため、ログやJavaプロセスの終了コードで実行結果の成否や状態を判断できるようにします。Spring Batchは、アプリケーション内で発生した例外やビジネスロジックの返却した処理結果から、バッチ処理の実行結果をJobExecutionやStepExecutionとして記録し、返却する終了コードを決定します。バッチ処理の実行結果と例外のスタックトレースはExitStatusクラスに格納され、JobExecutionやStepExecutionの中で管理されます。

■アプリケーション内で発生した例外から実行結果を決定

Stepの具象クラスが、アプリケーション内で発生した例外を検知します。 例外が発生しなかった場合は、正常終了を示すCOMPLETEDを、例外を検知した場合は異常終了を示すFAILEDと例外のスタックトレースをStepExecution内のExitStatusへ格納します。例外のスタックトレースはSpring Batchのログで確認できますが、永続化されているJobExecutionやStepExecution内のExitStatusでも確認することができます。

▶ チャンク方式におけるItemProcessorでの実装例

```
public class MyItemProcessor implements
        ItemProcessor<MyInputObject, MyOutputObject> {
    @Override
    public MyOutputObject process(MyInputObject item) throws Exception {
        // ・・・

        // データの状態が期待外であった場合
        throw new IllegalStateException("data is incorrect: " + item);

        // ・・・
    }
}
```

■ビジネスロジックの返却した処理結果から実行結果を決定

タスクレット方式の場合は、ビジネスロジックがメソッドの戻り値として処理結果の成否を返却できます。たとえば処理が正常終了した場合はCOMPLETEDを格納したExitStatusを、異常終了であることを示す場合はFAILEDを格納したExitStatusを返却するようにします。ExitStatusが返却されると、その内容がStepExecution内のExitStatusへ自動的に反映されます。

▶ タスクレット方式におけるビジネスロジックでの実装例

```
public class MyService {
    public ExitStatus execute() {
        // ・・・

        // 異常終了させる場合
        return ExitStatus.FAILED;
```

```
        // 正常終了の場合
        return ExitStatus.COMPLETED;
    }
}
```


CommandLineJobRunner でバッチ処理をコマンドラインから実行している場合は、ExitStatus を Java プロセスの終了コード（数値）へ変換します。終了コードへのデフォルトの変換ルールは SimpleJvmExitCodeMapper に定義されており以下のようになっています（**表 15.12**）。ExitCodeMapper を独自実装することで、変換ルールをカスタマイズすることが可能です。

表 15.12　ExitStatus の終了コードへのデフォルト変換ルール

ExitStatus 名	変換後の終了コード	意味
COMPLETED	0	正常終了
FAILED	1	異常終了
JOB_NOT_PROVIDED NO_SUCH_JOB	2	異常終了（ジョブが存在しない）
その他	1	異常終了

ExitStatus は、あらかじめ用意されている ExitStatus.COMPLETED や ExitStatus.FAILED 以外にも、任意の名前の ExitStatus を返却することが可能です。任意の名前に応じた終了コードへ変換することで、後続の処理を選択しわけたり、継続か終了かを判断したりと柔軟な制御が可能になります。

15.3.6 複数 Step のフロー制御

Job は複数の Step を連続して実行することができますが、その実行順序、実行条件を指定できます。バッチ処理を複数の Step に分割して実装することにより、ロジックの複雑化を低減しメンテナンス性を向上させたり、再利用性を高めたりすることができます。また、ジョブスケジューラを利用している場合は、ジョブスケジューラ側の管理を簡易化することにもつながるほか、Java プロセスの起動コスト削減効果が期待できます。

複数の Step の実行順序を定義する場合は Job に対して設定を行います。以下に挙げているように、Step を連続的に実行するための機能が Spring Batch にはあります。

- **逐次実行**
  複数の Step を、指定した順序で逐次実行します。前の Step の処理が完了してから次の Step が実行されます。ただし Step が異常終了した場合は、次の Step は実行されずに Job は処理を中断します。
- **条件付き実行**
  Step の ExitStatus などの条件に応じて次に実行する Step を選択するといった、条件分岐を含めることができます。

- **Job の終了**
  Job の終了を明示的に指定することができます。なお、Job の終了を明示的に指定しなくても、次に実行すべき Step が見つからなかった場合に Job は終了します。

以下に、複数 Step を連続実行させるための設定例を紹介します（**図 15.8**）。

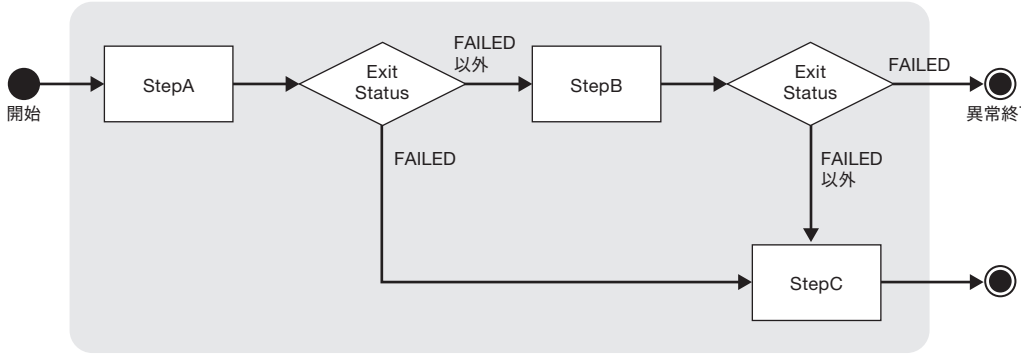

図 15.8　Step の実行を ExitStatus により条件分岐させる例

▶ **XML による Bean 定義例**

```
<batch:job id="job">
    <batch:step id="stepA" parent="s1">
        <batch:next on="*" to="stepB" />
        <batch:next on="FAILED" to="stepC" />
    </batch:step>
    <batch:step id="stepB" parent="s2">
        <batch:next on="*" to="stepC" />
        <batch:fail on="FAILED" /> <!-- 異常終了させることを明示的に定義 -->
    </batch:step>
    <batch:step id="stepC" parent="s3" />
</batch:job>
```

15.3.7 処理の並列実行

バッチ処理では、大量データに対して処理スループットを向上させるために、処理対象のデータを分割してマルチプロセスまたはマルチスレッドで処理を並列実行する場合があります。Spring Batch には、Step をマルチスレッドで並列実行する機能があります。Step を並列実行する方式として、split と partition の 2 つがあります（**図 15.9**）。

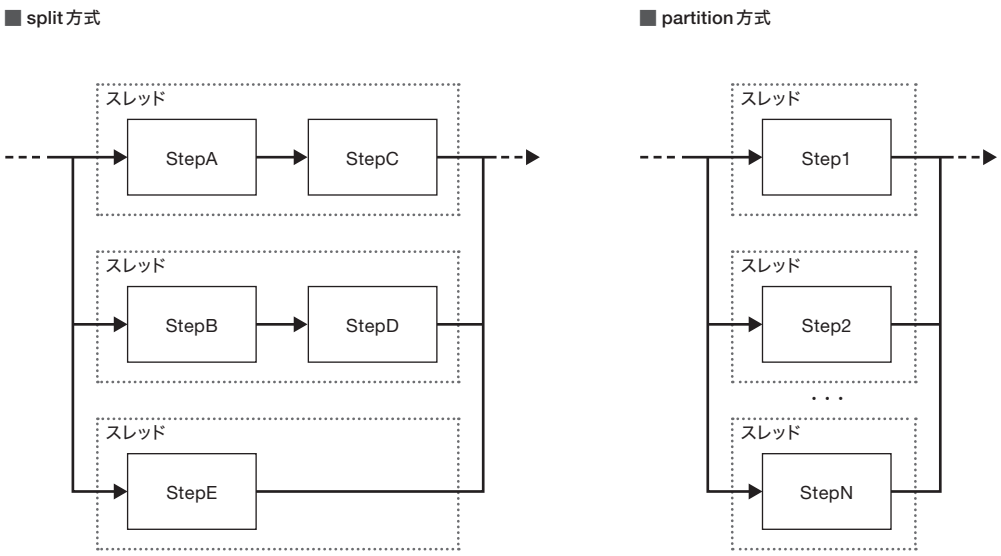

図 15.9 split方式と partition方式

■split方式

split 方式は、並列実行を行いたい Step を 1 つずつ個別に指定することが可能です。たとえば**図 15.14** の StepA-StepC、StepB-StepD、StepE のように、並行実行させる各スレッドで異なる Step を指定したり、1 つのスレッドの中で複数の Step を連続実行させるようなことが可能です。このように並列実行させる Step を細かく指定できる反面、並列数が多かったり可変だったりすると、スレッド数のチューニングや異常系への対処などの難易度が上がります。そのため、ある処理の裏で独立した別の処理を実行したいといった場合に適した方式となります。split 方式の場合、いずれかのスレッドで Step が異常終了した場合、他のスレッドで実行されている Step へは影響がありませんが、並列実行が完了した後の Step は実行されず Job が異常終了します。

▶ **XMLによるBean定義例**

```
<bean id="taskExecutor" class="org.springframework.core.task.SimpleAsyncTaskExecutor"/>

<batch:job id="job">
    <batch:split id="split1" task-executor="taskExecutor"
        next="stepA">
        <batch:flow>
            <batch:step id="stepA" parent="s1" next="stepC" />
            <batch:step id="stepC" parent="s3" />
        </batch:flow>
        <batch:flow>
            <batch:step id="stepB" parent="s2" next="stepD" />
            <batch:step id="stepD" parent="s4" />
        </batch:flow>
```

```
        <batch:flow>
            <batch:step id="stepE" parent="s5" />
        </batch:flow>
    </batch:split>
</batch:job>
```

■partition方式

partition方式は、特定の1つのStepを並列で実行させることができます。split方式のように、並列実行させるStepをスレッドごとに個別に指定できませんが、並列させる数はgrid-size属性で自由に調整できます。そのため、大量データを分割して並列処理させるような場合に適した方式となります。split方式の場合と同様、いずれかのスレッドでStepが異常終了した場合、他のスレッドで実行されているStepへは影響がありませんが、並列実行が完了した後のStepは実行されずJobが異常終了します。

▶ **XMLによるBean定義例**

```
<bean id="taskExecutor" class="org.springframework.core.task.SimpleAsyncTaskExecutor" />
<bean id="partitioner" class= "org.springframework.batch.core.partition.support.SimplePartitioner" />

<batch:job id="job">
    <batch:step id="step1.master">
        <batch:partition step="step1" partitioner="partitioner">
            <batch:handler grid-size="10" task-executor="taskExecutor" />
        </batch:partition>
    </batch:step>
</batch:job>
```

並列実行されるStepがまったく同一の内容で処理をしてしまうと、Step同士が同じデータを処理してしまいます。そこで各Stepに対して処理対象データを割り当てる必要があります。Partitionerインターフェイスを使用すると、並列実行される各Stepに対しユニークとなる実行名を割り当てたり、StepExecutionに登録するExecutionContextをスレッドごとに生成し、割り当てることができます。たとえば、ユニークな番号を採番してExecutionContextに格納するPartitioner具象クラスを実装し、ItemReaderでそのユニークな番号を含めたSQLを発行することで、各スレッドのStepの入力データが競合しないようにすることができます。

▶ **Partitionerの実装例**

```
public class MyPartitioner implements Partitioner {

    public static final String PARTITION_NO_PROPERTY = "partitionNo";

    private static final String PARTITION_KEY = "partition";

    @Override
    public Map<String, ExecutionContext> partition(int gridSize) {

        Map<String, ExecutionContext> map = new HashMap<String, ExecutionContext>();
```

```
        for (int i = 0; i < gridSize; ++i) {
            ExecutionContext executionContext = new ExecutionContext();
            // ExecutionContextにパーティション番号を格納
            executionContext.putInt(PARTITION_NO_PROPERTY, i);
            map.put(PARTITION_KEY + i, executionContext);
        }
        return map;
    }
}
```


いずれの方式を選択した場合にも、マルチスレッドで同一のStepが並列実行される場合はItemReaderなどのStepに含まれる各構成要素がスレッドセーフであるか、stepスコープのBeanである必要があります。

■ 執筆者紹介

倉元 貴一（くらもと きいち）［Spring Batchの章担当］
株式会社 NTT データ
入社後数年間、Springを用いたオープン系システム開発を経て、金融機関のミッションクリティカルシステム向けJavaフレームワークの整備／導入支援に従事。近年著者陣に合流し、TERASOLUNA Batch Frameworkのリーダーをしつつ、普及展開活動にも参加している。オープン系の技術進化をキャッチアップしつつ、子どもの進化に驚く日々。