

# What's new in Spring Batch 5

JSUG SpringFest 2023/3/17  
JSUGスタッフ 池谷 智行

# 自己紹介

- ・日本Springユーザ会(JSUG)スタッフ
- ・某Slirでソフトウェアアーキテクト  
(実態は管理業務多め・・・)

- ・某書の筆者の一人

<https://www.shoeisha.co.jp/book/detail/9784798142470>



# Spring Batchの過去の勉強会

JSUG勉強会@2017/5/25



<https://www.slideshare.net/apkiban/ss-122881217>

JSUG勉強会@2020/5/13

知っておきたい  
Spring Batch Tips

JSUG 勉強会 2020/5/13  
池谷 智行

Batchは作ったことあるけど、  
初めてSpring Batchを使ってみた

JSUG勉強会 2020年その4 Spring Batch  
2020-05-13 (水)

へー／heisy (笹倉 秀行)

<https://www.slideshare.net/ikeyat/spring-batch-tips-233698138>  
<https://www.slideshare.net/HideyukiSASAKURA/batch-spring-batch>

Spring Fest 2020@12/17



<https://www.slideshare.net/nttdata-tech/spring-fest-2020-spring-batch-nttdata>

今回：Spring Batchのおさらい+Spring Batch 5の新機能など

# アジェンダ

- Spring Batchとは？
  - Tasklet v.s. Chunk
- Spring Batch5
  - Spring Batch4からの変更
  - 新機能

# アジェンダ

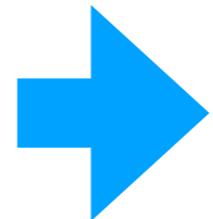
- Spring Batchとは？
  - Tasklet v.s. Chunk
- Spring Batch5
  - Spring Batch4からの変更
  - 新機能

# バッチ処理とは？

一般的に、バッチ処理とは「まとめて一括処理する」ことを指す。

## バッチ処理を採用する目的

- ・スループットの向上
- ・（オンライン処理の）応答性の確保
- ・時間やイベントへの対応
- ・外部システムとの連携上の制約



## 求められる要件

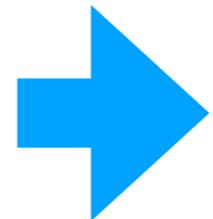
- ・性能向上
- ・異常発生時のリカバリ
- ・多様な起動方式
- ・さまざまな入出力インターフェース

# バッチ処理とは？

一般的に、バッチ処理とは「まとめて一括処理する」ことを指す。

## バッチ処理を採用する目的

- ・スループットの向上
- ・（オンライン処理の）応答性の確保
- ・時間やイベントへの対応
- ・外部システムとの連携上の制約



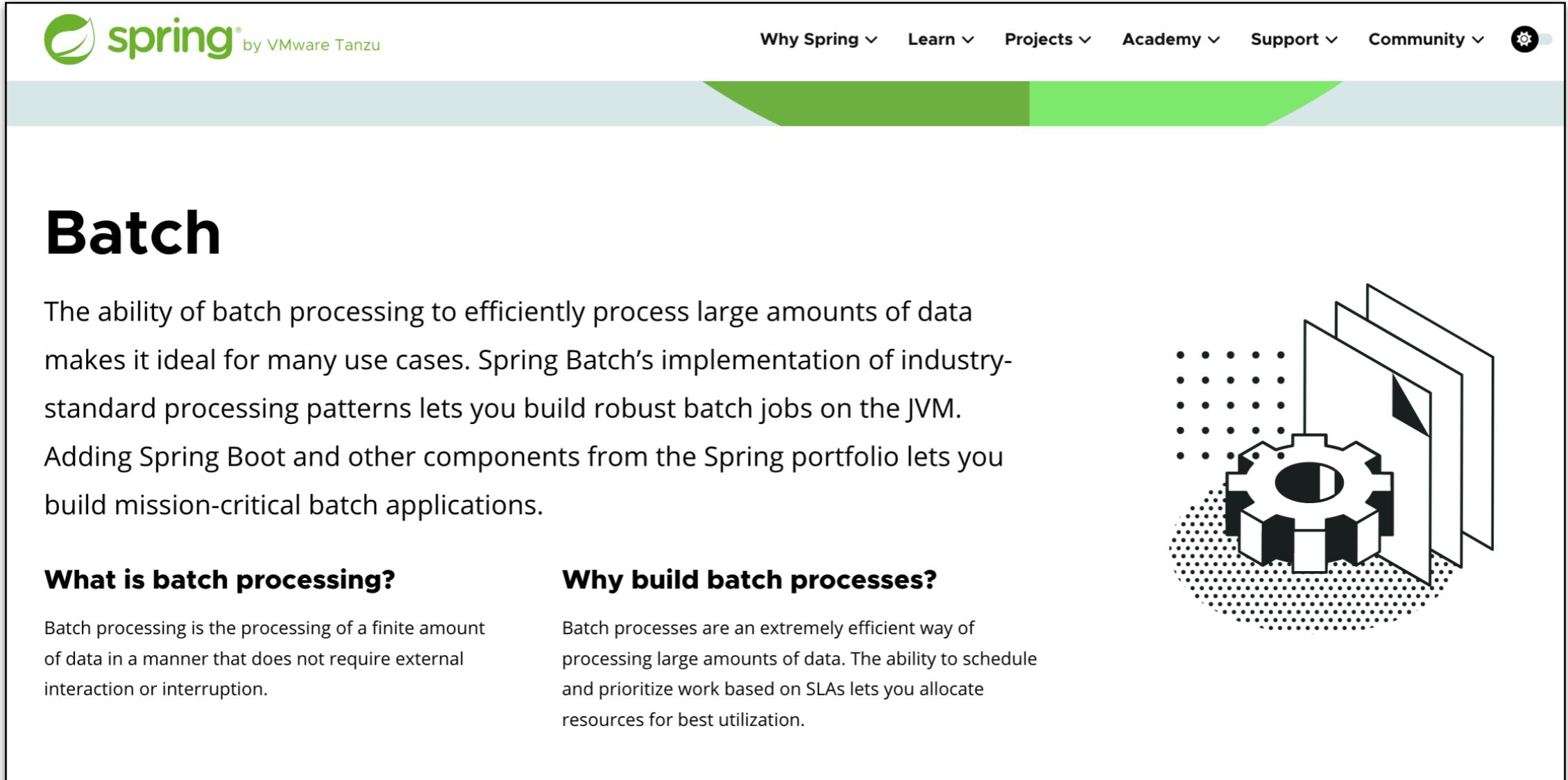
## 求められる要件

- ・性能向上
- ・異常発生時のリカバリ
- ・多様な起動方式
- ・さまざまな入出力インターフェース

**Spring Batch  
がサポート**

# Spring Batchとは？

Springフレームワークをベースとした、  
バッチ処理向けJavaアプリケーションフレームワーク



The screenshot shows the official Spring website at <https://spring.io>. The top navigation bar includes links for Why Spring, Learn, Projects, Academy, Support, Community, and a user icon. The main content area features a large title "Batch" in bold black font. Below it is a descriptive paragraph about the efficiency of batch processing for large amounts of data. To the right is a graphic of a gear and a stack of documents.

**Batch**

The ability of batch processing to efficiently process large amounts of data makes it ideal for many use cases. Spring Batch's implementation of industry-standard processing patterns lets you build robust batch jobs on the JVM. Adding Spring Boot and other components from the Spring portfolio lets you build mission-critical batch applications.

**What is batch processing?**

Batch processing is the processing of a finite amount of data in a manner that does not require external interaction or interruption.

**Why build batch processes?**

Batch processes are an extremely efficient way of processing large amounts of data. The ability to schedule and prioritize work based on SLAs lets you allocate resources for best utilization.

<https://spring.io/batch>

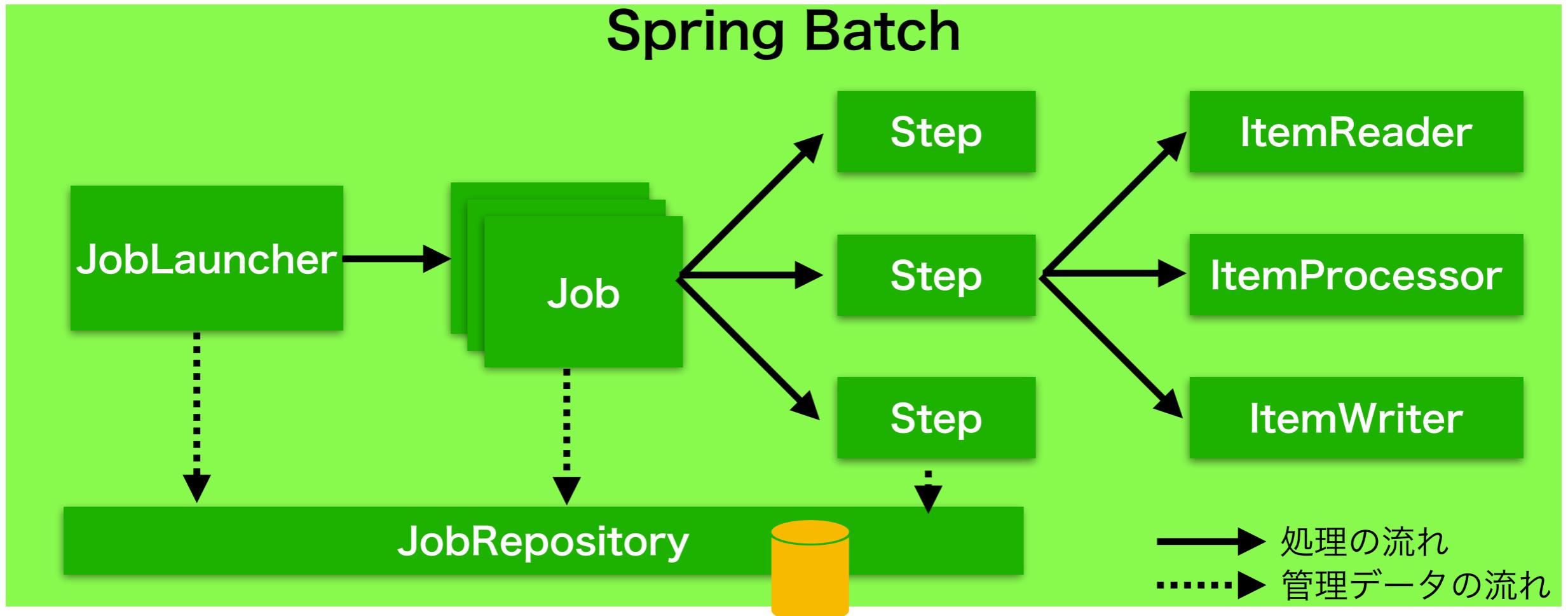
# Spring Batchとは？

## 提供される代表的な機能

- 処理の流れを定型化する機能（タスクレット、チャンク）
- 様々な起動方法（コマンドライン、Servlet等）
- 様々なデータ形式の入出力（ファイル、DB、キュー、等）
- 処理の効率化（多重実行、並列実行、条件分岐、等）
- ジョブの管理（実行状況の永続化、リスタート、等）

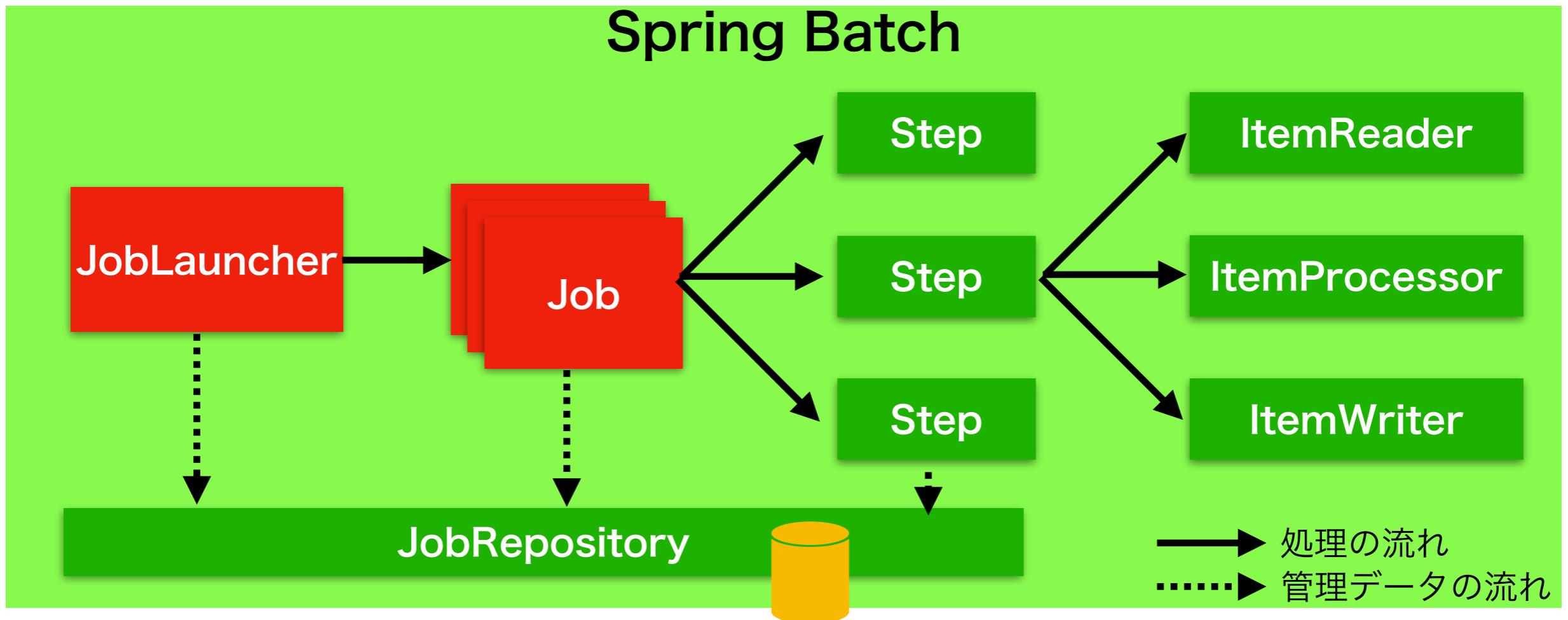
[https://terasoluna-batch.github.io/guideline/current/ja/single\\_index.html#Ch02\\_SpringBatchArch\\_Overview](https://terasoluna-batch.github.io/guideline/current/ja/single_index.html#Ch02_SpringBatchArch_Overview)

# Spring Batchの基本構成



<https://docs.spring.io/spring-batch/docs/5.0.0/reference/html/job.html#configureJob>

# Spring Batchの基本構成



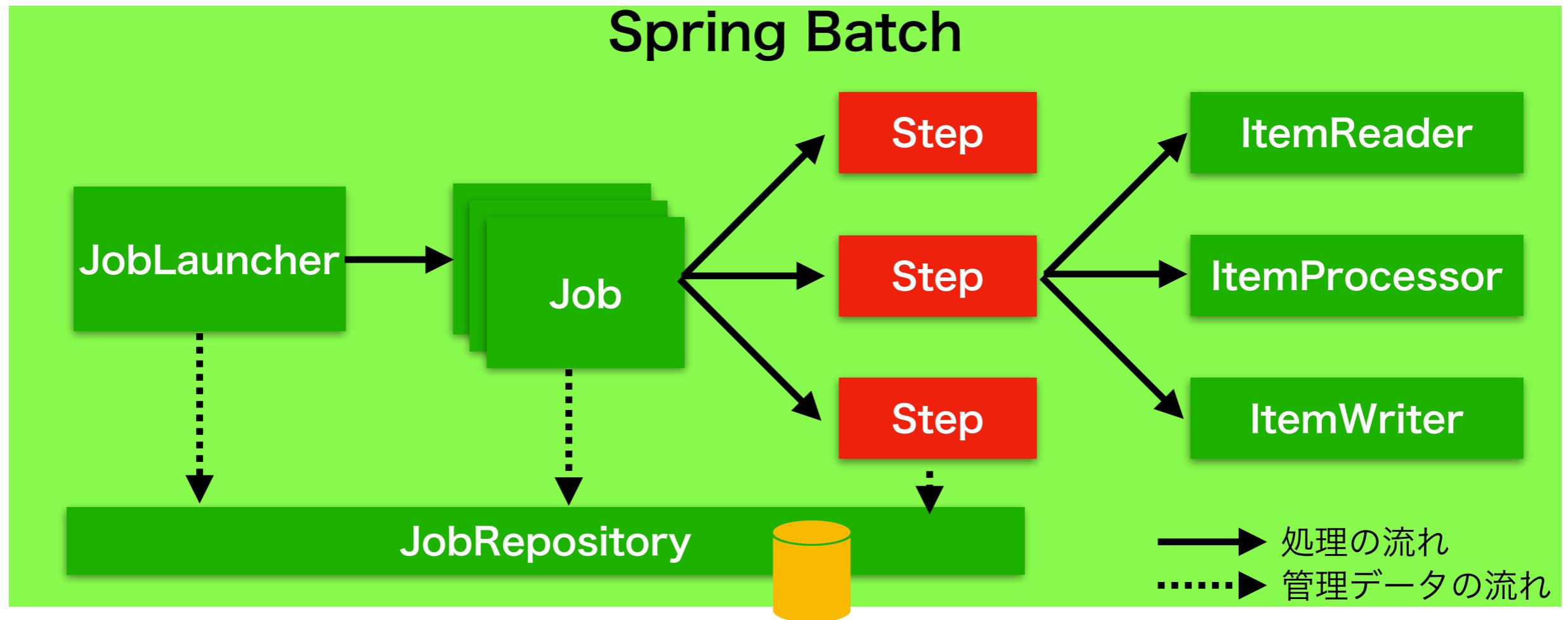
## JobLauncher

**Job**を起動するインターフェイス。例えば、コマンドラインからの起動時に**Job**を起動するために呼ばれる。

## Job

バッチ処理の一連をまとめた**1実行単位**。実際の処理は**Step**に分割して実装し呼び出す。

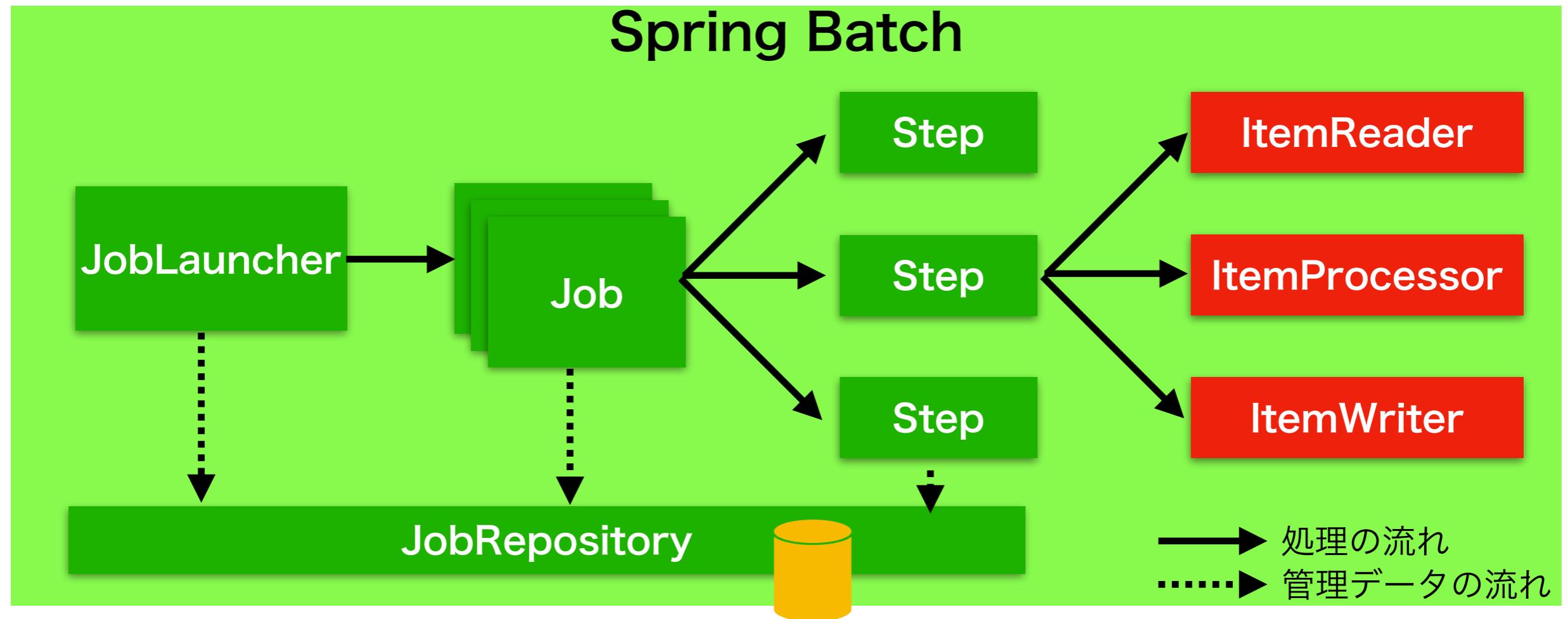
# Spring Batchの基本構成



## Step

処理の実装や実行管理を行う最小単位。1つのJobを複数のStepに分割して処理することにより、処理の再利用、並列化、条件分岐が可能になる。チャンクモデルまたはタスクレットモデルのいずれかで実装する。

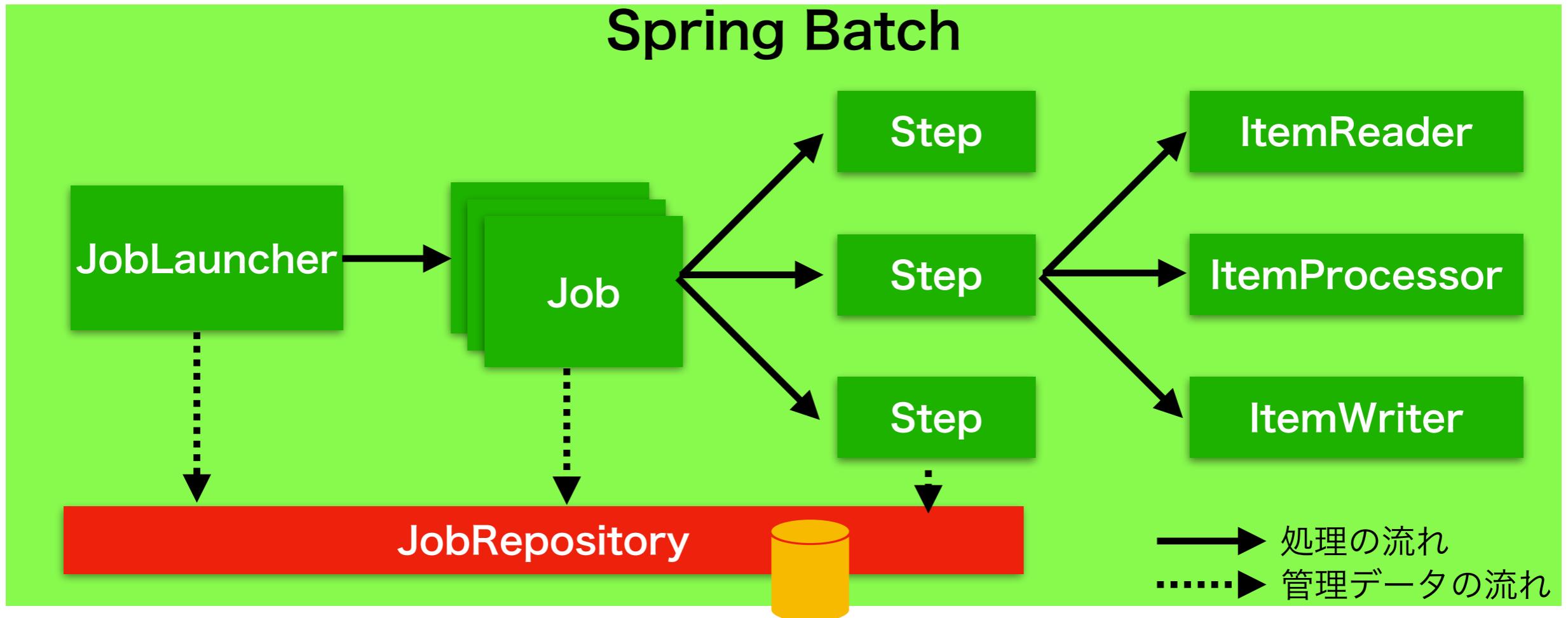
# Spring Batchの基本構成



## ItemReader/ItemProcessor/ItemWriter

チャンクモデルを実装する際に、データの入力/加工/出力の3つに分割するためのインターフェース。自分で実装するか、Spring Batchが提供する実装(DB、ファイルの入出力処理など)を利用する。

# Spring Batchの基本構成



## JobRepository

JobやStepの実行状態、結果を外部DBで管理するためのインターフェイス  
業務ロジックとは別トランザクションでDBアクセス。  
(永続化したくない場合は、H2等のインメモリDBに格納させ揮発させる)

# 参考：JobRepositoryのテーブル構造

Job内のStepの状態・実行結果  
(Step実行の度に増加)

BATCH_STEP_EXECUTION	
○	STEP_EXECUTION_ID : BIGINT[20]
○	VERSION : BIGINT[20]
○	STEP_NAME : VARCHAR[100]
□	JOB_EXECUTION_ID : BIGINT[20]
	START_TIME : DATETIME
	END_TIME : DATETIME
	STATUS : VARCHAR[10]
	COMMIT_COUNT : BIGINT[20]
	READ_COUNT : BIGINT[20]
	FILTER_COUNT : BIGINT[20]
	WRITE_COUNT : BIGINT[20]
	READ_SKIP_COUNT : BIGINT[20]
	WRITE_SKIP_COUNT : BIGINT[20]
	PROCESS_COUNT : BIGINT[20]
	ROLLBACK_COUNT : BIGINT[20]
	EXIT_CODE : VARCHAR[100]
	EXIT_MESSAGE : VARCHAR[2500]
	LAST_UPDATE : DATETIME

BATCH_JOB_INSTANCE	
○	JOB_INSTANCE_ID : BIGINT[20]
○	VERSION : BIGINT[20]
○	JOB_NAME : VARCHAR[100]
○	JOB_KEY : VARCHAR[32]

Job (Job名×Job引数の単位で  
増加)

BATCH_JOB_EXECUTION	
○	JOB_EXECUTION_ID : BIGINT[20]
○	VERSION : BIGINT[20]
○	JOB_INSTANCE_ID : BIGINT[20]
○	CREATE_TIME : DATETIME
○	START_TIME : DATETIME
○	END_TIME : DATETIME
○	STATUS : VARCHAR[10]
○	EXIT_CODE : VARCHAR[100]
○	EXIT_MESSAGE : VARCHAR[2500]
○	LAST_UPDATE : DATETIME

Jobの状態・実行結果  
(Job実行の度に増加)

BATCH_JOB_EXECUTION_PARAMS	
□	JOB_EXECUTION_ID : BIGINT[20]
○	TYPE_CD : VARCHAR[6]
○	KEY_NAME : VARCHAR[100]
○	STRING_VAL : VARCHAR[100]
○	DATE_VAL : DATETIME
○	LONG_VAL : BIGINT[20]
○	DOUBLE_VAL : DOUBLE
○	IDENTIFYING : CHAR[1]

Job引数

BATCH_STEP_EXECUTION_CONTEXT	
□	STEP_EXECUTION_ID : BIGINT[20]
○	SHORT_CONTEXT : VARCHAR[2500]
○	SERIALIZE_CONTEXT : TEXT

Stepコンテキストオブジェクトの  
シリアルライズデータ

[https://terasoluna-batch.github.io/guideline/current/ja/single\\_index.html#Ch02\\_SpringBatchArch\\_Arch\\_MetadataSchema](https://terasoluna-batch.github.io/guideline/current/ja/single_index.html#Ch02_SpringBatchArch_Arch_MetadataSchema)

# Hello World

※Spring Boot利用前提

## Step1 : Spring InitializrでSpring Batchプロジェクトを生成



Project

Gradle - Groovy  Java  Kotlin  Groovy

Gradle - Kotlin  Maven

Spring Boot

好みでMaven

3.1.0 (SNAPSHOT)  3.1.0 (M1)  3.0.5 (SNAPSHOT)

3.0.4  2.7.10 (SNAPSHOT)  2.7.9

### Project Metadata

Group com.example

Artifact demo-batch-v5

Name demo-batch-v5

Description Demo project for Spring Boot

Package name com.example.demo-batch-v5

### Dependencies

ADD DEPENDENCIES... ⌘ + B

#### Spring Batch I/O

Batch applications with transactions, retry/skip and chunk based processing.

#### H2 Database SQL

Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Spring Batch  
H2 Database  
をDependencyに追加

GENERATE ⌘ + ↵

EXPLORE CTRL + SPACE

SHARE...

# Hello World

※Spring Boot利用前提

## Step2 : Spring Batchのstarterが適用されていることを確認

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-batch</artifactId>
  </dependency>

  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>
...
```

# Hello World

※Spring Boot利用前提

## Step3：ジョブパラメータ（コマンド引数）をログに出力するタスクレットを実装

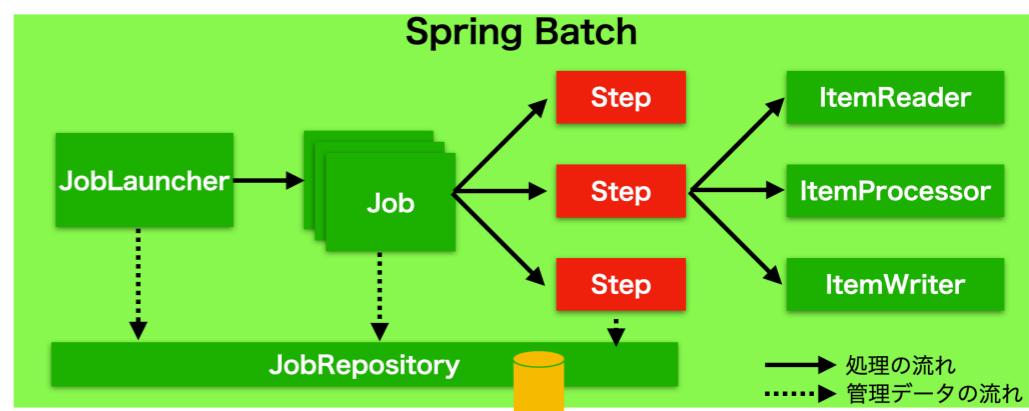
```
@Bean  
@StepScope  
public Tasklet demoTakslet() {  
    return new Tasklet() {  
        @Override  
        public RepeatStatus execute(StepContribution stepContribution,  
                                    ChunkContext chunkContext) throws Exception {  
  
            StepExecution stepExecution = stepContribution.getStepExecution();  
            JobParameters jobParameters = stepExecution.getJobParameters();  
            String firstName = jobParameters.getString("firstName");  
            String lastName = jobParameters.getString("lastName");  
  
            log.info("Hello World, {} {}", lastName, firstName);  
  
            return RepeatStatus.FINISHED;  
        }  
    };  
}
```

# Hello World

※Spring Boot利用前提

## Step3：ジョブパラメータ（コマンド引数）をログに出力するタスクレットを実装

```
@Bean  
@StepScope  
public Tasklet demoTasklet() { Taskletインターフェイスを実装しBean登録  
    return new Tasklet() {  
        @Override  
        public RepeatStatus execute(StepContribution stepContribution,  
            ChunkContext chunkContext) throws Exception {  
  
            StepExecution stepExecution = stepContribution.getStepExecution();  
            JobParameters jobParameters = stepExecution.getJobParameters();  
            String firstName = jobParameters.getString("firstName");  
            String lastName = jobParameters.getString("lastName");  
  
            log.info("Hello World, {} {}", lastName, firstName);  
  
            return RepeatStatus.FINISHED;  
        }  
    };  
}
```



### Step

処理の実装や実行管理を行う最小単位。1つのJobを複数のStepに分割して処理することにより、処理の再利用、並列化、条件分岐が可能になる。チャンクモデルまたはタスクレットモデルのいずれかで実装する。

# Hello World

※Spring Boot利用前提

## Step3：ジョブパラメータ（コマンド引数）をログに出力するタスクレットを実装

```
@Bean  
@StepScope  
public Tasklet demoTakslet() {  
    return new Tasklet() {  
        @Override  
        public RepeatStatus execute(StepContribution stepContribution,  
                                    ChunkContext chunkContext) throws Exception {  
  
            StepExecution stepExecution = stepContribution.getStepExecution();  
            JobParameters jobParameters = stepExecution.getJobParameters();  
            String firstName = jobParameters.getString("firstName");  
            String lastName = jobParameters.getString("lastName");  
  
            log.info("Hello World, {} {}", lastName, firstName);  
  
            return RepeatStatus.FINISHED;  
        }  
    };  
}
```

ジョブパラメータ  
を取得

ログにジョブパラメータを出力

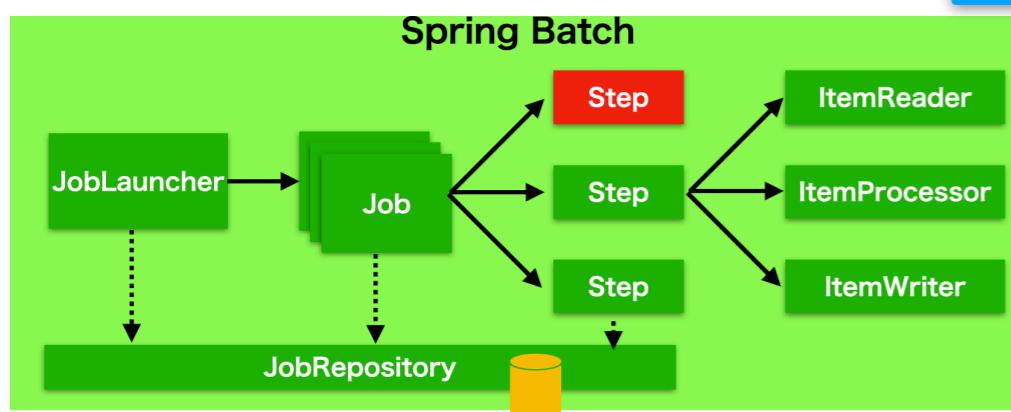
# Hello World

※Spring Boot利用前提

## Step4 : Stepを作成・登録し、Step3のタスクレットを設定する

```
@Bean  
public Step demoStep(JobRepository jobRepository,  
PlatformTransactionManager transactionManager, Tasklet tasklet) {  
  
    return new StepBuilder("demoStepTasklet", jobRepository)  
        .tasklet(tasklet, transactionManager)  
        .build();  
}
```

前述Taskletを呼び出すStepを作成



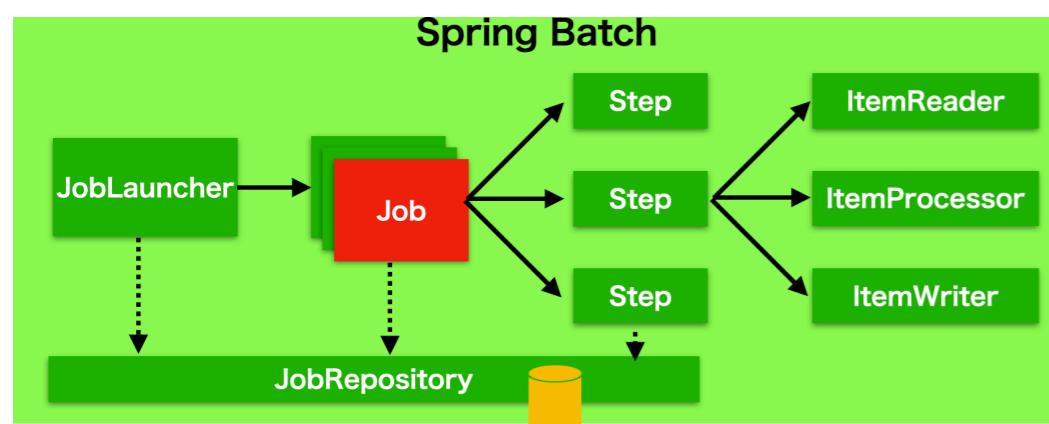
# Hello World

※Spring Boot利用前提

## Step5：Jobを作成・登録し、Step4のStepを設定する

```
@Bean  
public Job demoJob(JobRepository jobRepository, Step step) {  
    return new JobBuilder("demoJob", jobRepository)  
        .start(step)  
        .build();  
}
```

前述Stepを呼び出すJobを作成



# Hello World

## ※Spring Boot利用前提

## Step6：アプリケーションを実行する（コマンド引数なし）

ログは出たが、ジョブパラメータ（コマンド引数）未指定のためnull

# Hello World

## ※Spring Boot利用前提

## Step7：アプリケーションを実行する（コマンド引数あり）

```
$ mvn package
```

1

```
$ java -jar target/demo-batch-v5-0.0.1-SNAPSHOT.jar firstName=TARO  
lastName=SATO
```

2023-03-11T02:11:51.087+09:00 INFO 63302 --- [c.e.demobatchv5.DemoBatchV5Application : Hello World. SATO TARO

ジョブパラメータが正しく表示された

# アジェンダ

- Spring Batchとは？
  - Tasklet v.s. Chunk
- Spring Batch5
  - Spring Batch4からの変更
  - 新機能

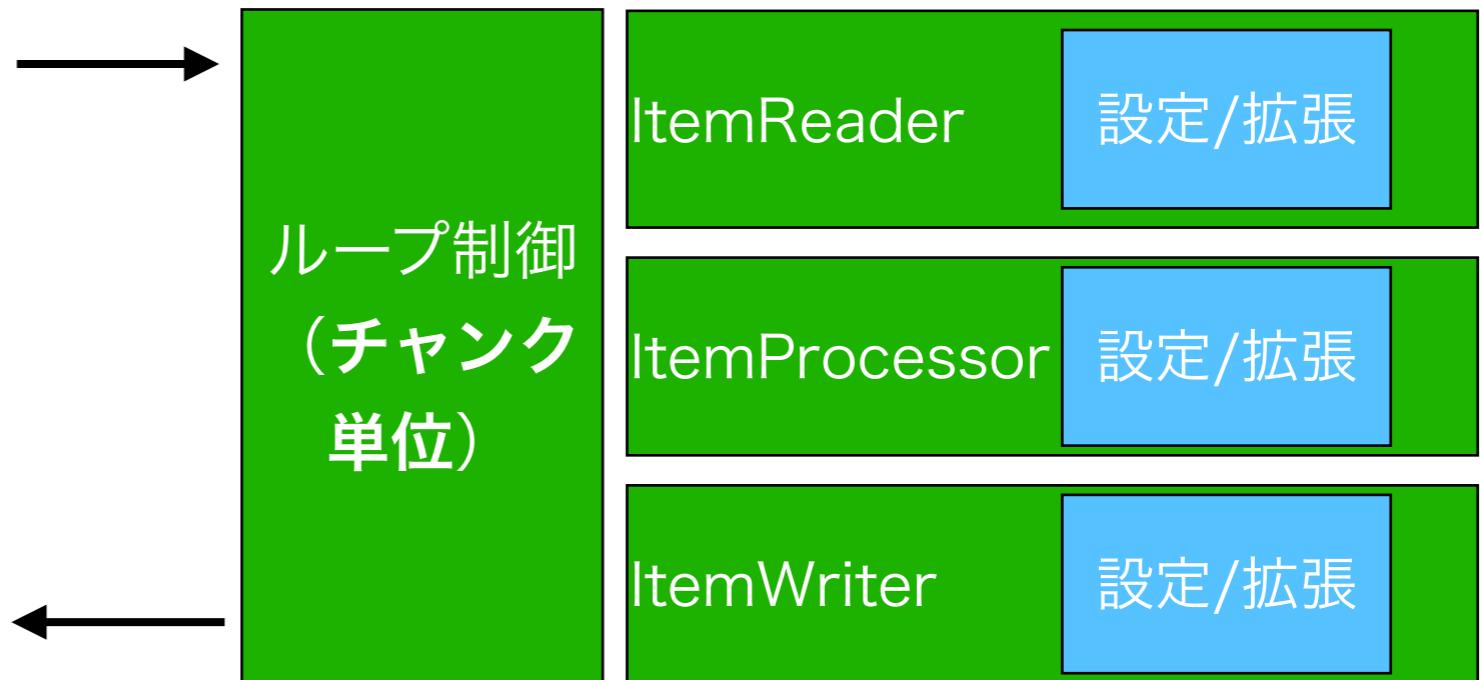
# Taskletとは？Chunkとは？

ざっくりの違い

Taskletモデル  
業務ロジック



Chunkモデル  
業務ロジック



# Tasklet (タスクレット) モデルの実装例

```
@Bean  
@StepScope  
public Tasklet demoTakslet() {  
    return new Tasklet() {  
        @Override  
        public RepeatStatus execute(StepContribution stepContribution,  
            ChunkContext chunkContext) throws Exception {  
  
            // ビジネスロジックを実装  
  
            return RepeatStatus.FINISHED;  
        }  
    };  
}
```

Taskletインターフェイスを実装しBean登録

```
@Bean  
public Step demoStep(JobRepository jobRepository,  
PlatformTransactionManager transactionManager, Tasklet tasklet) {  
  
    return new StepBuilder("demoStepTasklet", jobRepository)  
        .tasklet(tasklet, transactionManager)  
        .build();  
}
```

Tasklet BeanをStepに登録

# Chunk (チャンク) モデルの実装例(1/2)

```
@Bean  
@StepScope  
public FlatFileItemReader fileItemReader(  
    @Value("#{jobParameters['inputFile']}") String filePath) {  
    // 中略  
  
    FlatFileItemReader<InputHoge> fileItemReader  
        = new FlatFileItemReader<>();  
    fileItemReader.setResource(new FileSystemResource(filePath));  
    fileItemReader.setLineMapper(lineMapper);  
    return fileItemReader;  
}  
  
①ItemReaderのBean定義  
(例ではCSVファイル入力)  
  
@Bean  
@StepScope  
public ItemProcessor itemProcessor() {  
    return new ItemProcessor<InputHoge, OutputFuga>() {  
  
        @Override  
        public OutputFuga process(InputHoge inputHoge)  
            throws Exception {  
            // 中略(入力InputHoge>出力OutputFugaの変換ロジック)  
            return outputFuga;  
        }  
    };  
}  
②ItemProcessorのBean定義  
Spring Batch提供クラスに設定適用
```

# Chunk (チャンク) モデルの実装例(1/2)

```
@Bean  
@StepScope  
public FlatFileItemReader fileItemReader(  
    @Value("#{jobParameters['inputFile']}") String filePath) {  
    // 中略  
    FlatFileItemReader<InputHoge> fileItemReader  
        = new FlatFileItemReader<>();  
    fileItemReader.setResource(new FileSystemResource(filePath));  
    fileItemReader.setLineMapper(lineMapper);  
    return fileItemReader;  
}  
  
@Bean  
@StepScope  
public ItemProcessor itemProcessor() {  
    return new ItemProcessor<InputHoge, OutputFuga>() {  
        @Override  
        public OutputFuga process(InputHoge inputHoge)  
            throws Exception {  
            // 中略(入力InputHoge>出力OutputFugaの変換ロジック)  
            return outputFuga;  
        }  
    };  
}
```

加工前 (ItemProcessorへのoutput)

加工前 (ItemReaderからのinput)

加工後 (ItemWriterへのoutput)

# Chunk (チャンク) モデルの実装例(2/2)

```
@Bean  
@StepScope  
public FlatFileItemWriter fileItemWriter(  
    @Value("#{jobParameters['outputFile']}") String filePath) {  
  
    // 中略  
  
    FlatFileItemWriter<OutputFuga> fileItemWriter  
        = new FlatFileItemWriter<>();  
    fileItemWriter.setResource(new FileSystemResource(filePath));  
    fileItemWriter.setLineAggregator(aggregator);  
  
    return fileItemWriter;  
}
```

③WriterのBean定義  
(例ではCSVファイル出力)

Spring Batch提供クラスに設定適用

```
@Bean  
public Step demoStepChunk(JobRepository jobRepository, PlatformTransactionManager  
transactionManager, ItemReader<InputHoge> reader, ItemProcessor<InputHoge,  
OutputFuga> processor, ItemWriter<OutputFuga> writer) {  
    return new StepBuilder("demoChunk", jobRepository)  
        .<InputHoge, OutputFuga>chunk(10, transactionManager)  
        .reader(reader)  
        .processor(processor)  
        .writer(writer)  
        .build();  
}
```

④ItemReader/ItemProcessor/ItemWriter  
各BeanをStepに登録

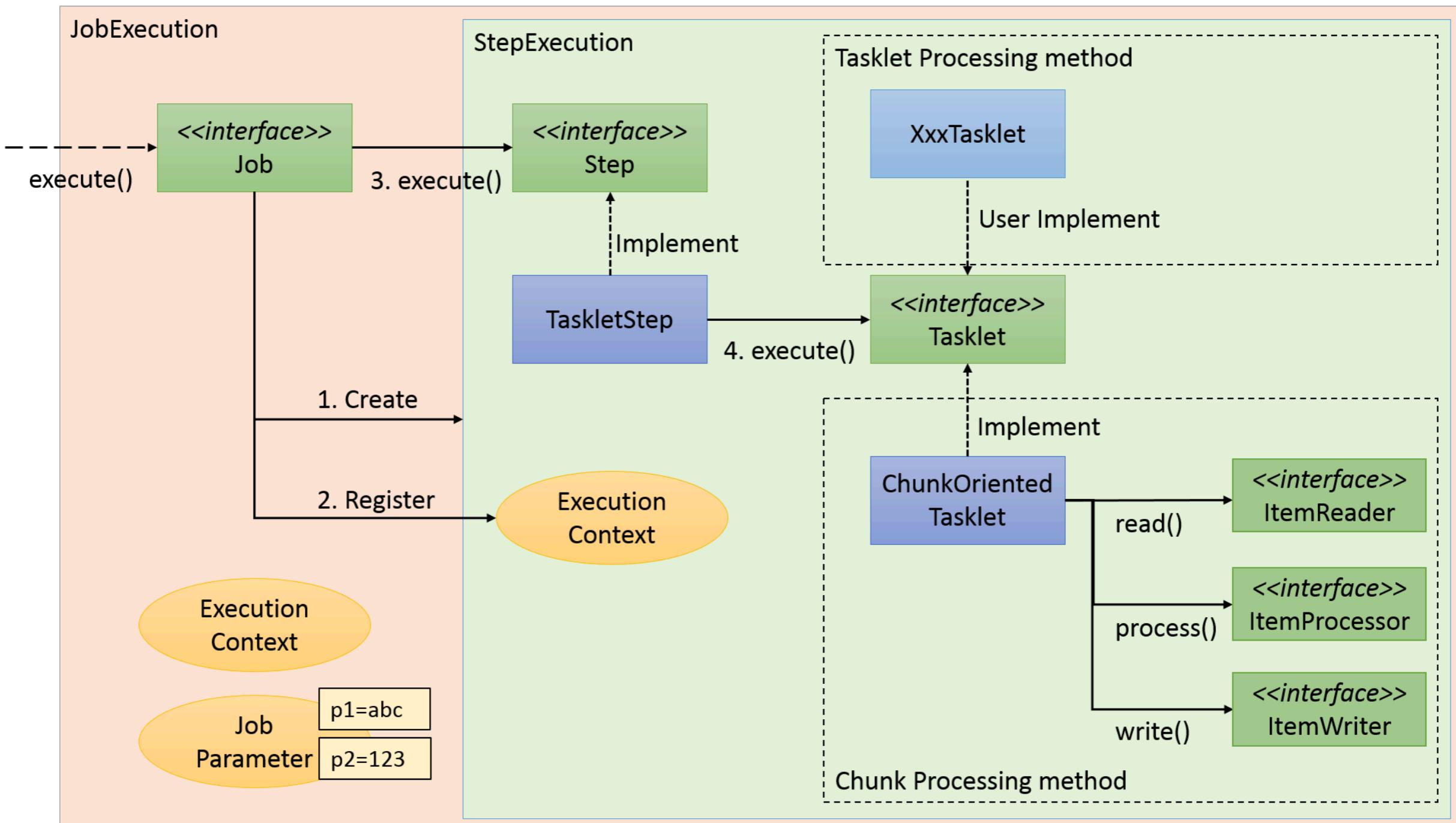
チャンクサイズ  
設定

# 参考：代表的なReader/Processor/Writer

	ItemReader	ItemProcessor	ItemWriter
ファイル	<b>FlatFileItemReader</b> StaxEventItemReader JsonFileItemReader	-	<b>FlatFileItemWriter</b> StaxEventItemWriter JsonFileItemWriter
	JdbcCursorItemReader JdbcPagingItemReader MyBatisCursorItemReader MyBatisPagingItemReader JpaPagingItemReader HibernateCursorItemReader HibernatePagingItemReader	-	JdbcBatchItemWriter  MyBatisBatchItemWriter  JpaItemWriter HibernateItemWriter
その他	MongoItemReader JmsItemReader AmqpItemReader	PassThroughItemProcessor ValidatingItemProcessor CompositeItemProcessor	JmsItemWriter AmqpItemWriter

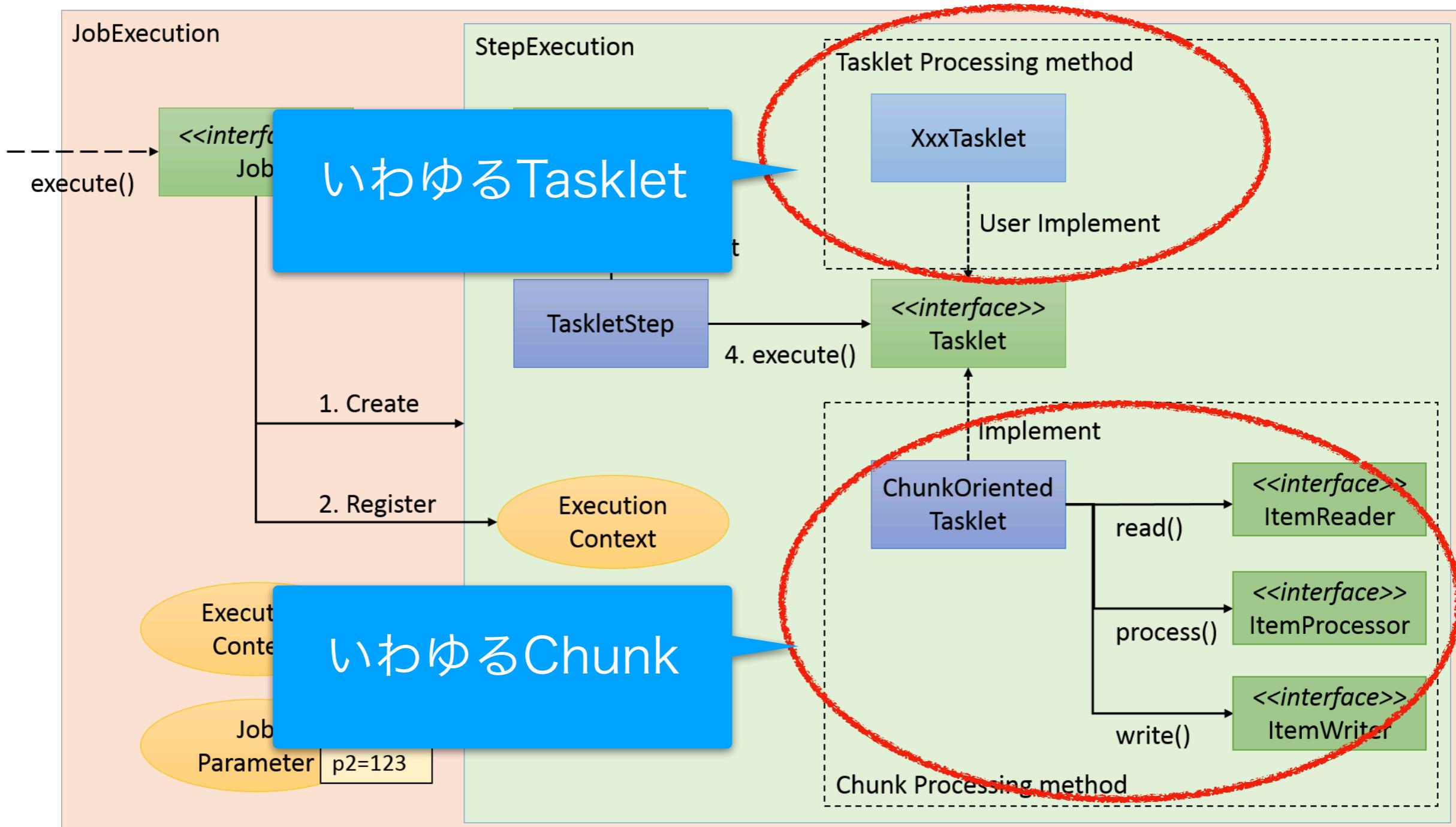
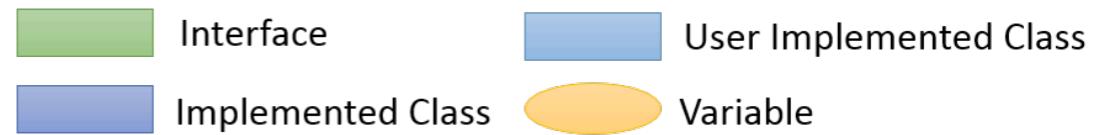
# 参考：TaskletとChunkの関係

## Spring Batch全体像



# 参考：TaskletとChunkの関係

## Spring Batch全体像



# 特徴

観点	Taskletモデル	Chunkモデル	考察
業務ロジック記述の自由度	自由度高い (ロジック全てを開発者が記述)	自由度低い (Reader、Processor、Writerの組み合わせ)	Chunkは設計時点で構造意識が必要で、Taskletよりも学習コスト高い。
トランザクション	1トランザクション	件数／チャンクサイズに分割	リランでリカバリしたい場合はTasklet、リスタートでリカバリや大量データ処理にはChunk。
リストア	未対応	対応	

[https://terasoluna-batch.github.io/guideline/current/ja/single\\_index.html#Ch03\\_ChunkOrTasklet](https://terasoluna-batch.github.io/guideline/current/ja/single_index.html#Ch03_ChunkOrTasklet)

# アジェンダ

- Spring Batchとは?
  - Tasklet v.s. Chunk
- Spring Batch5
  - Spring Batch4からの変更
  - 新機能（ネイティブサポート、Observability）

# Spring Batch5

- Spring6世代のSpring Batch
- Spring6のGraalVMネイティブビルドやObservabilityに追従
- Spring Batch4からの**非互換性は少しあり**

	Spring 5.3 Spring Batch 4.3 Spring Boot 2.6/2.7	Spring 6.0 Spring Batch 5.0 Spring Boot 3.0
Java	8~17 (or 18)	17~
Java EE Jakarta EE	7~8 (Tomcat 8.5/9, Jetty)	9~ <b>(Tomcat 10, Jetty 11)</b>
Hibernate	5	6

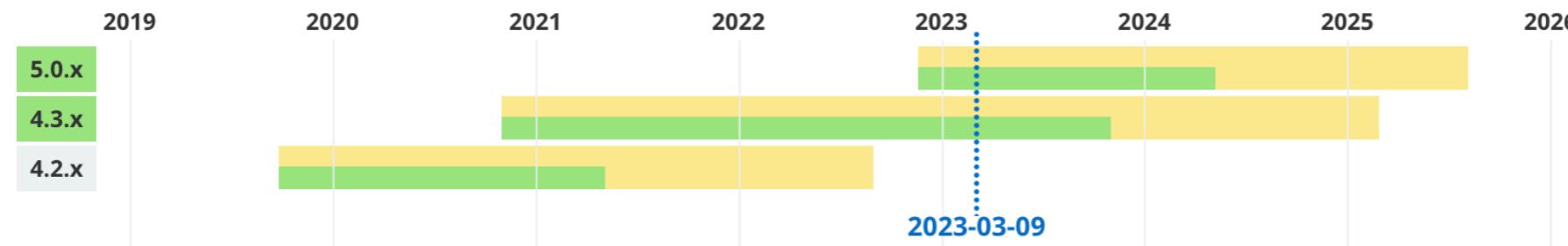
# Spring Batch5のサポート期間

Spring Batch 5.0.1



OVERVIEW LEARN SUPPORT SAMPLES

Branch	Initial Release	End of Support	End Commercial Support *
5.0.x	2022-11-23	2024-05-18	2025-08-18
4.3.x	2020-10-28	2023-11-18	2025-02-18
4.2.x	2019-10-02	2021-05-20	2022-09-20



## OSS support

Free security updates and bugfixes with support from the Spring community. See [VMware Tanzu OSS support policy](#).

## Commercial support

Business support from Spring experts during the OSS timeline, plus extended support after OSS End-Of-Life.

Publicly available releases for critical bugfixes and security issues when requested by customers.

## Future release

Generation not yet released, timeline is subject to changes.

# アジェンダ

- Spring Batchとは?
  - Tasklet v.s. Chunk
- Spring Batch5
  - Spring Batch4からの変更
  - 新機能

# Spring Batch4からの変更（抜粋）

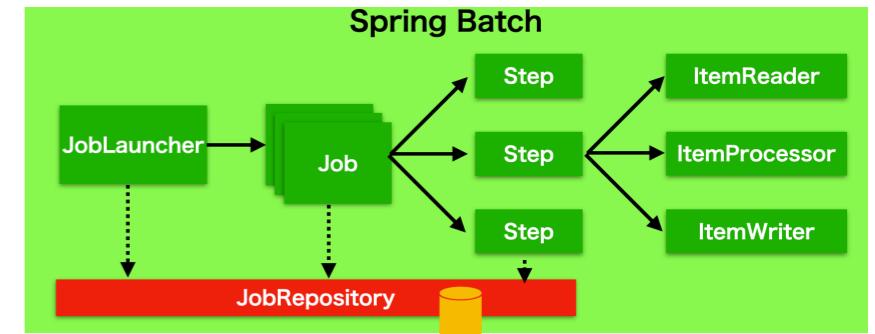
- Java17がベースライン
- MapベースJobRepositoryの廃止
- トランザクションマネージャBeanが非公開
- @EnableBatchProcessingに属性追加
- JobやStepのBean作成方法の変更
- ジョブパラメータの型、指定形式の変更
- JSR-352(jBatch) 実装の削除
- DBスキーマの変更
- Java records(Java16+)への対応
- Spring Bootで@EnableBatchProcessingが不要
- DefaultBatchConfiguration
- Spring Bootの複数Job実行機能の廃止

<https://docs.spring.io/spring-batch/docs/5.0.0/reference/html/whatsnew.html#whatsNew>

<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Migration-Guide#spring-batch-changes>

# Spring Batch4からの変更（抜粋）

- Java17がベースライン
- MapベースJobRepositoryの廃止
- トランザクションマネージャBeanが非公開
- @EnableBatchProcessingに属性追加
- JobやStepのBean作成方法の変更
- ジョブパラメータの型、指定形式の変更
- JSR-352(jBatch) 実装の削除
- DBスキーマの変更
- Java records(Java16+)への対応
- Spring Bootで@EnableBatchProcessingが不要
- DefaultBatchConfiguration
- Spring Bootの複数Job実行機能の廃止



<https://docs.spring.io/spring-batch/docs/5.0.0/reference/html/whatsnew.html#whatsNew>

<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Migration-Guide#spring-batch-changes>

# Spring Batch4からの変更（抜粋）

- Java17がベースライン
- MapベースJobRepositoryの廃止
- **トランザクションマネージャBeanが非公開**
- @EnableBatchProcessingに属性追加
- JobやStepのBean作成方法の変更
- ジョブパラメータの型、指定形式の変更
- JSR-352(jBatch) 実装の削除
- DBスキーマの変更
- Java records(Java16+)への対応
- Spring Bootで@EnableBatchProcessingが不要
- DefaultBatchConfiguration
- Spring Bootの複数Job実行機能の廃止

JobRepository用のトランザクションマネージャがアプリケーションで誤用されるのを防ぐ。

<https://docs.spring.io/spring-batch/docs/5.0.0/reference/html/whatsnew.html#whatsNew>  
<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Migration-Guide#spring-batch-changes>

# Spring Batch4からの変更（抜粋）

- Java17がベースライン
- MapベースJobRepositoryの廃止
- トランザクションマネージャBeanが非公開
- ~~@EnableBatchProcessingに属性追加~~
- **JobやStepのBean作成方法の変更**
- ジョブパラメータの型、指定形式の変更
- JSR-352(jBatch) 実装の削除
- DBスキーマの変更
- Java records(Java16+)への対応
- **Spring Bootで@EnableBatchProcessingが不要**
- **DefaultBatchConfiguration**
- **Spring Bootの複数Job実行機能の廃止**

JobRepositoryが使うJDBC  
データソースやトランザクショ  
ンマネージャをカスタム可能

<https://docs.spring.io/spring-batch/docs/5.0.0/reference/html/whatsnew.html#whatsNew>  
<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Migration-Guide#spring-batch-changes>

# Spring Batch4からの変更（抜粋）

- Java17がベースライン
- MapベースJobRepositoryの廃止
- トランザクションマネージャBeanが非公開
- @EnableBatchProcessingに属性追加
- **JobやStepのBean作成方法の変更**
- ジョブパラメータの型、指定形式の変更
- JSR-352(jBatch) 実装の削除
- DBスキーマの変更
- Java records(Java16+)への対応
- **Spring Bootで@EnableBatchProcessingが不要**
- **DefaultBatchConfiguration**
- **Spring Bootの複数Job実行機能の廃止**

<https://docs.spring.io/spring-batch/docs/5.0.0/reference/html/whatsnew.html#whatsNew>

<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Migration-Guide#spring-batch-changes>

# JobやStepのBean作成方法の変更

- JobBuilderFactory、StepBuilderFactoryが非推奨化された
- 各FactoryBeanも自動的に作成されなくなった

## JobのBean作成方法

```
@Bean  
public Job demoJob(Step step) {  
    return jobBuilderFactory.get("demoJob")  
        .start(step)  
        .build();  
}
```

V4

```
@Bean  
public Job demoJob(JobRepository jobRepository, Step step) {  
    return new JobBuilder("demoJob", jobRepository)  
        .start(step)  
        .build();  
}
```

V5

<https://github.com/spring-projects/spring-batch/wiki/Spring-Batch-5.0-Migration-Guide#jobbuilderfactory-and-stepbuilderfactory-bean-exposureconfiguration>

※JobRepositoryが内部で生成されていることが分かりづらくミスリードするため改善された。

※非推奨ではあるが、自前で本FactoryをBean化すればSpring Batch5でも利用継続可能

<https://github.com/spring-projects/spring-batch/issues/4188>

# JobやStepのBean作成方法の変更

## StepのBean作成方法

```
@Bean  
public Step demo01Step(Tasklet tasklet) {  
    return stepBuilderFactory.get("demoTasklet")  
        .tasklet(tasklet)  
        .build();  
}
```

V4

```
@Bean  
public Step demo01Step(JobRepository jobRepository,  
    PlatformTransactionManager tm, Tasklet tasklet) {  
    return new StepBuilder("demoTasklet", jobRepository)  
        .tasklet(tasklet, tm)  
        .build();  
}
```

V5

※JobRepository用のトランザクションマネージャBeanが非公開化されたため、  
Stepで使用するトランザクションマネージャを明示的に指定する必要あり

<https://github.com/spring-projects/spring-batch/wiki/Spring-Batch-5.0-Migration-Guide#transaction-manager-bean-exposureconfiguration>

# Spring Batch4からの変更（抜粋）

- Java17がベースライン
- MapベースJobRepositoryの廃止
- トランザクションマネージャBeanが非公開
- @EnableBatchProcessingに属性追加
- JobやStepのBean作成方法の変更
- ジョブパラメータの型、指定形式の変更
- JSR-352(jBatch) 実装の削除
- DBスキーマの変更
- Java records(Java16+)への対応
- Spring Bootで@EnableBatchProcessingが不要
- DefaultBatchConfiguration
- Spring Bootの複数Job実行機能の廃止

<https://docs.spring.io/spring-batch/docs/5.0.0/reference/html/whatsnew.html#whatsNew>

<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Migration-Guide#spring-batch-changes>

# ジョブパラメータの型、指定形式の変更

- ジョブパラメータ（起動引数）に使える型の制約が緩和された
- ジョブパラメータの指定形式が型の改善に合わせて見直された

[+|-]パラメータ名(パラメータの型)=パラメータの値

V4

使用可能な型(parameterType)はstring, long, double, dateのみ

パラメータ名=パラメータの値,パラメータの型,identificationFlag

V5

or

```
パラメータ名='{"value": "パラメータの値", "type": "パラメータの型",  
"identifying": "boolean値"}'
```

- 型(parameterType)は、**fully qualified**な型名（例：java.time.LocalDate）、省略時はString。使用可能な型はConversionServiceにより拡張可能。
- identificationFlagは、ジョブインスタンスを特定するキーに含めるか否か（省略時はtrue）

<https://docs.spring.io/spring-batch/docs/5.0.0/reference/html/whatsnew.html#support-for-any-type-as-a-job-parameter>

# Spring Batch4からの変更（抜粋）

- Java17がベースライン
- MapベースJobRepositoryの廃止
- トランザクションマネージャBeanが非公開
- @EnableBatchProcessingに属性追加
- JobやStepのBean作成方法の変更
- ジョブパラメータの型、指定形式の変更
- JSR-352(jBatch) 実装の削除  
    あまり使われなかったため  
(+Jakarta EE移植?)
- DBスキーマの変更
- Java records(Java16+)への対応
- Spring Bootで@EnableBatchProcessingが不要
- DefaultBatchConfiguration
- Spring Bootの複数Job実行機能の廃止

<https://docs.spring.io/spring-batch/docs/5.0.0/reference/html/whatsnew.html#whatsNew>

<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Migration-Guide#spring-batch-changes>

# Spring Batch4からの変更（抜粋）

- Java17がベースライン
- MapベースJobRepositoryの廃止
- トランザクションマネージャBeanが非公開
- @EnableBatchProcessingに属性追加
- JobやStepのBean作成方法の変更
- ジョブパラメータの型、指定形式の変更
- JSR-352(jBatch) 実装の削除
- **DBスキーマの変更**
- Java records(Java16+)への対応
- Spring Bootで@EnableBatchProcessingが不要
- DefaultBatchConfiguration
- Spring Bootの複数Job実行機能の廃止

<https://docs.spring.io/spring-batch/docs/5.0.0/reference/html/whatsnew.html#whatsNew>

<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Migration-Guide#spring-batch-changes>

# DBスキーマの変更

- JobRepositoryが利用するDBテーブルの構成が変更された
- Spring Batch4からアップデートする場合は、テーブルのマイグレーションが必要

テーブル	V4→V5の変更
BATCH_JOB_EXECUTION	以下カラムの削除 (JSR-352削除) <ul style="list-style-type: none"><li>JOB_CONFIGURATION_LOCATION</li></ul>
BATCH_JOB_EXECUTION_PARAMS	以下カラムの削除 <ul style="list-style-type: none"><li>TYPE_CD</li><li>KEY_NAME</li><li>STRING_VAL</li><li>DATE_VAL</li><li>LONG_VAL</li><li>DOUBLE_VAL</li></ul> 以下カラムの追加 <ul style="list-style-type: none"><li>PARAMETER_NAME</li><li>PARAMETER_TYPE</li><li>PARAMETER_VALUE</li></ul> (ジョブパラメータの型改善)

各DB製品向けのマイグレーション用DDLは、spring-batch-core.jarの以下に配置されている。  
[org/springframework/batch/core/migration/5.0](https://github.com/spring-projects/spring-batch/wiki/Spring-Batch-5.0-Migration-Guide#database-schema-updates)

<https://github.com/spring-projects/spring-batch/wiki/Spring-Batch-5.0-Migration-Guide#database-schema-updates>

# Spring Batch4からの変更（抜粋）

- Java17がベースライン
- MapベースJobRepositoryの廃止
- トランザクションマネージャBeanが非公開
- @EnableBatchProcessingに属性追加
- JobやStepのBean作成方法の変更
- ジョブパラメータの型、指定形式の変更
- JSR-352(jBatch) 実装の削除
- DBスキーマの変更
- Java records(Java16+)への対応
- Spring Bootで@EnableBatchProcessingが不要
- DefaultBatchConfiguration
- Spring Bootの複数Job実行機能の廃止

ItemReaderでrecordsを生成

<https://docs.spring.io/spring-batch/docs/5.0.0/reference/html/whatsnew.html#whatsNew>  
<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Migration-Guide#spring-batch-changes>

# Spring Batch4からの変更（抜粋）

- Java17がベースライン
- MapベースJobRepositoryの廃止
- トランザクションマネージャBeanが非公開
- @EnableBatchProcessingに属性追加
- JobやStepのBean作成方法の変更
- ジョブパラメータの型、指定形式の変更
- JSR-352(jBatch) 実装の削除
- DBスキーマの変更
- Java records(Java16+)への対応
- **Spring Bootで@EnableBatchProcessingが不要**
- DefaultBatchConfiguration
- Spring Bootの複数Job実行機能の廃止

<https://docs.spring.io/spring-batch/docs/5.0.0/reference/html/whatsnew.html#whatsNew>

<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Migration-Guide#spring-batch-changes>

# Spring Bootで@EnableBatchProcessingが不要

- Spring Batch4までは@EnableXxxが必要だったが、不要になった。
- 付与してしまうとJobが起動しない事象が発生するので注意

<https://github.com/spring-projects/spring-batch/issues/4251>

```
@SpringBootApplication  
@EnableBatchProcessing  
public class BatchDemo01Application {  
...}
```

V4

```
@SpringBootApplication  
//@EnableBatchProcessing  
public class BatchDemo01Application {  
...}
```

V5

※新設されたDefaultBatchConfigurationの拡張により、Auto-Configの上書き設定が可能。

<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Migration-Guide#enablebatchprocessing-is-now-discouraged>

# Spring Batch4からの変更（抜粋）

- Java17がベースライン
- MapベースJobRepositoryの廃止
- トランザクションマネージャBeanが非公開
- @EnableBatchProcessingに属性追加
- JobやStepのBean作成方法の変更
- ジョブパラメータの型、指定形式の変更
- JSR-352(jBatch) 実装の削除
- DBスキーマの変更
- Java records(Java16+)への対応
- Spring Bootで@EnableBatchProcessingが不要
- **DefaultBatchConfiguration**
- **Spring Bootの複数Job実行機能の廃止**

<https://docs.spring.io/spring-batch/docs/5.0.0/reference/html/whatsnew.html#whatsNew>

<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Migration-Guide#spring-batch-changes>

# DefaultBatchConfiguration

- @EnableBatchProcessing以外の新たなSpring Batch初期化手段
- Spring BootのAuto-Configとの併用が可能

## 利用例&デフォルト設定を変更する例

```
@Configuration  
class MyBatchConfiguration extends DefaultBatchConfiguration {  
    @Override  
    protected Charset getCharset() {  
        // デフォルトUTF-8から変更  
        return StandardCharsets.ISO_8859_1;  
    }  
}
```

V5

※類似した機能をこれまで提供していたBatchConfigurerは廃止

<https://docs.spring.io/spring-batch/docs/5.0.0/api/org/springframework/batch/core/configuration/support/DefaultBatchConfiguration.html>

# Spring Batch4からの変更（抜粋）

- Java17がベースライン
- MapベースJobRepositoryの廃止
- トランザクションマネージャBeanが非公開
- @EnableBatchProcessingに属性追加
- JobやStepのBean作成方法の変更
- ジョブパラメータの型、指定形式の変更
- JSR-352(jBatch) 実装の削除
- DBスキーマの変更
- Java records(Java16+)への対応
- Spring Bootで@EnableBatchProcessingが不要
- DefaultBatchConfiguration
- Spring Bootの複数Job実行機能の廃止

<https://docs.spring.io/spring-batch/docs/5.0.0/reference/html/whatsnew.html#whatsNew>

<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Migration-Guide#spring-batch-changes>

# Spring Bootの複数Job実行機能の廃止

複数Jobを連続実行する機能が廃止された。

パラメータ名もnames→nameに変更されているため注意。

```
$ java -jar myapp.jar -Dspring.batch.job.names=<Job名※>
```

V4

```
<Job引数名1>=<値1> <Job引数名2>=<値2> ...
```

※カンマ区切で複数指定可能、省略時は全Jobが起動

```
$ java -jar myapp.jar -Dspring.batch.job.name=<Job名※>
```

V5

```
<Job引数名1>=<値1> <Job引数名2>=<値2> ...
```

※カンマ区切で複数指定不可、省略時はJobが1個である必要あり

<https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-3.0-Migration-Guide#spring-batch-changes>

# アジェンダ

- Spring Batchとは?
  - Tasklet v.s. Chunk
- Spring Batch5
  - Spring Batch4からの変更
  - 新機能

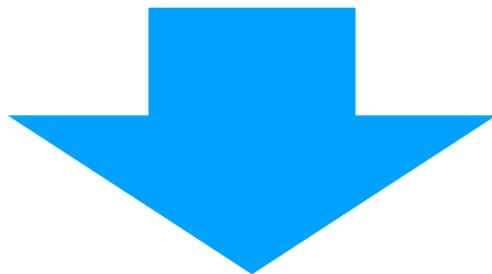
- ・新機能1：GraalVMネイティブサポート
- ・新機能2：Micrometerによるトレース収集

# 新機能1：GraalVMネイティブサポート

Spring6やSpring Boot3のGraalVMネイティブサポートにより、

**Spring Batch**アプリケーションもネイティブビルドに対応

(Spring6やSpring Boot3のネイティブサポート自体については割愛)



バッチアプリケーションとしての期待

- ・ **起動時間の短縮**（オンラインアプリケーションより効果が大きい）
- ・ バッチならではの**複雑計算処理の短縮**（例えば大量ソートなど）

さっそく検証してみる

# GraalVMネイティブサポート：起動時間の検証

ミニマムなバッチアプリケーションをネイティブビルドし、起動時間を比較してみる

## Step1：Spring InitializrでSpring Batchプロジェクトを生成

The screenshot shows the Spring Initializr interface with the following configurations:

- Project**: Maven (selected)
- Language**: Java (selected)
- Spring Boot**: 3.0.4 (selected)
- Project Metadata**:
  - Group: com.example
  - Artifact: demo-batch-native
  - Name: demo-batch-native
  - Description: Demo project for Spring Boot
- Dependencies**:
  - GraalVM Native Support** (Developer Tools): Support for compiling Spring applications to native executables using the GraalVM native-image compiler.
  - Spring Batch** (I/O): Batch applications with transactions, retry/skip and chunk based processing.
  - H2 Database** (SQL): Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Red boxes highlight the "Maven" selection, the "GraalVM Native Support" dependency, and the "Spring Batch" dependency. Red text annotations are overlaid on the right side of the dependencies section:

- GraalVM Native Support
- Spring Batch
- H2 Database
- をDependencyに追加

At the bottom, there are three buttons: GENERATE, EXPLORE, and SHARE... .

<https://start.spring.io/>

# GraalVMネイティブサポート：起動時間の検証

## Step2：空のTaskletを実装し、JobやStepに組み込む

```
@Bean  
public Tasklet demoTakslet() {  
    return new Tasklet() {  
        @Override  
        public RepeatStatus execute(StepContribution stepContribution,  
            ChunkContext chunkContext) throws Exception {  
            log.info("test tasklet 01.");  
            return RepeatStatus.FINISHED;  
        }  
    };  
}  
// 以下略
```

# GraalVMネイティブサポート：起動時間の検証

## Step3：JVMで実行し起動時間を確認

```
$ mvn package  
...  
  
$ java -jar target/demo-batch-native-0.0.1-SNAPSHOT.jar  
...  
  
Started DemoBatchNativeApplication in 2.706 seconds (process  
running for 3.398)  
...  
  
Step: [demoTasklet] executed in 17ms  
...  
  
Job: [SimpleJob: [name=demoJob]] completed with the following  
parameters: [{}] and the following status: [COMPLETED] in 43ms  
...
```

# GraalVMネイティブサポート：起動時間の検証

## Step4 : buildpacksでネイティブビルド (Docker必要)

```
$ mvn -Pnative spring-boot:build-image
...
[INFO] [creator] [1/7] Initializing... (15.3s @ 0.15GB)
[INFO] [creator] [2/7] Performing analysis... [*****] (221.6s @ 2.13GB)
[INFO] [creator] [3/7] Building universe... (34.7s @ 2.88GB)
[INFO] [creator] [4/7] Parsing methods... [****] (23.0s @ 1.97GB)
[INFO] [creator] [5/7] Inlining methods... [**] (14.5s @ 2.43GB)
[INFO] [creator] [6/7] Compiling methods... [*****] (177.1s @ 2.48GB)
[INFO] [creator] [7/7] Creating image... (16.4s @ 3.38GB)
...
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 10:17 min
[INFO] Finished at: 2023-03-08T22:39:04+09:00
[INFO] Final Memory: 32M/120M
[INFO] -----
```

(参考) Docker Desktopに3CPU/9GB割当@Macbook 2017

# GraalVMネイティブサポート：起動時間の検証

## Step5：ネイティブ実行し起動時間を確認

```
$ docker run --rm -p 8080:8080 docker.io/library/demo-batch-native:0.0.1-SNAPSHOT
```

...

```
Started DemoBatchNativeApplication in 0.046 seconds (process  
running for 0.069)
```

...

```
Step: [demoTasklet] executed in 1ms
```

...

```
Job: [SimpleJob: [name=demoJob]] completed with the following  
parameters: [{}] and the following status: [COMPLETED] in 3ms
```

...

# GraalVMネイティブサポート：起動時間の検証

## 比較

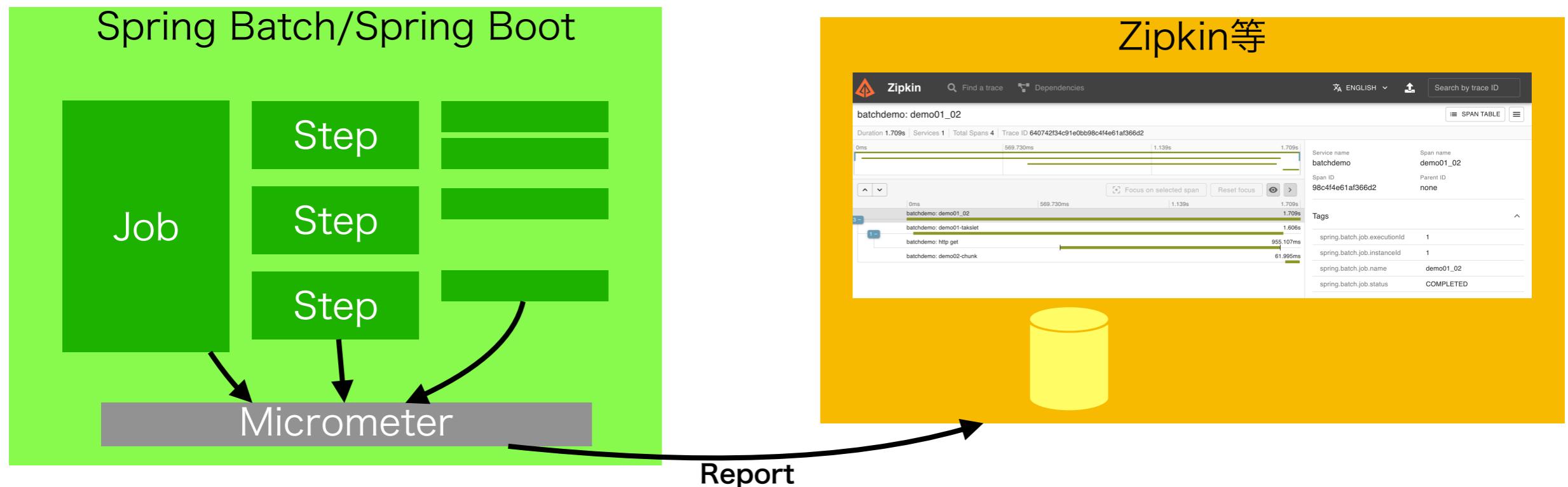
観点	JVM	ネイティブ
起動時間	3398ms	69ms
Step実行時間 (空Tasklet)	17ms	1ms
Job実行時間	43ms	3ms

起動時間の割合が多く占める、短命のバッチ処理の場合効果が大きい  
(例：短周期的に繰り返し起動される掃除バッチ／Pollingバッチなど)

- ・新機能1：GraalVMネイティブサポート
  - ・新機能2：Micrometerによるトレース収集
- 

# 新機能2：Micrometerによるトレース収集

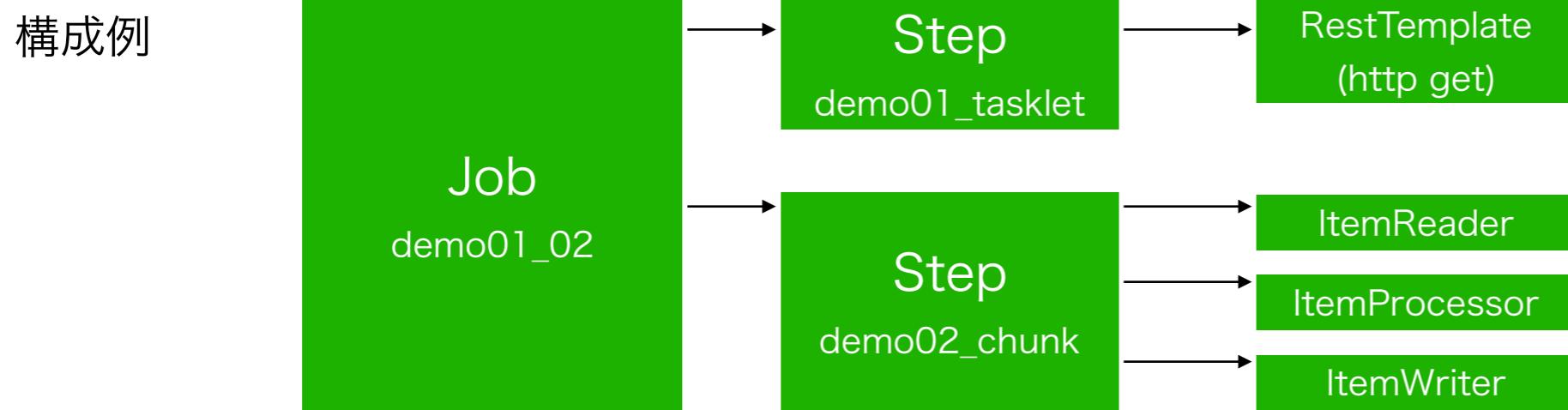
Spring6やMicrometer1.10のObservability機能の仕組みを用いた、  
バッチ処理のトレース収集に対応 (Job/Step)



バッチ処理の性能解析（ボトルネック分析）や  
キャパシティプランなどに活用可能

# 新機能2：Micrometerによるトレース収集

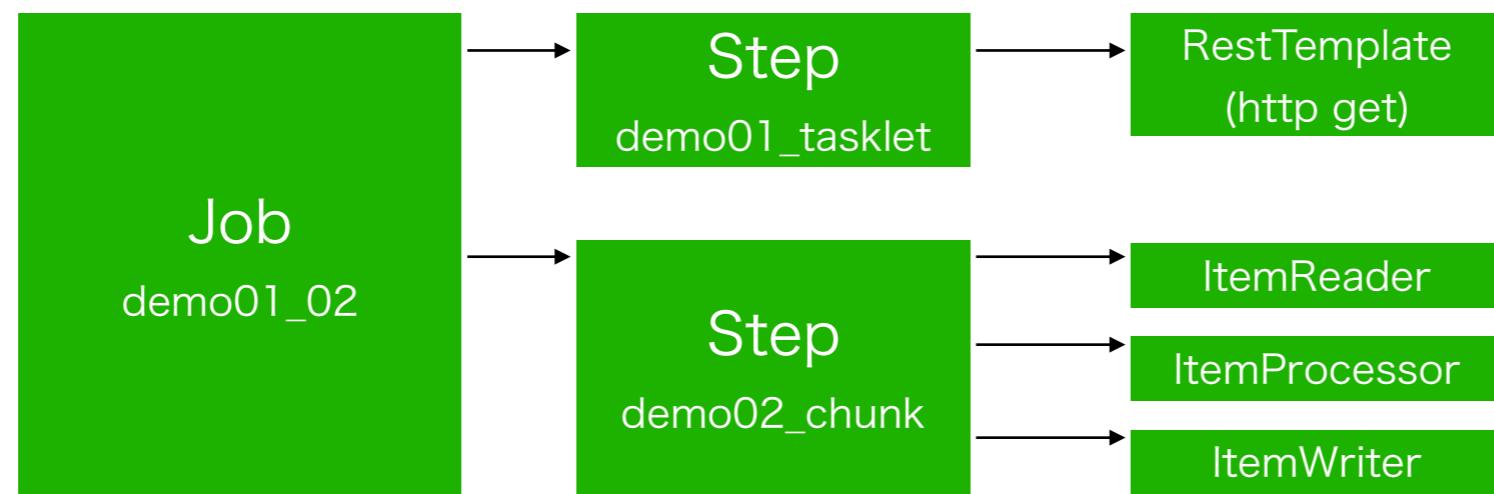
Job実行単位にトレースが切られ、Step単位にスパンが切られる



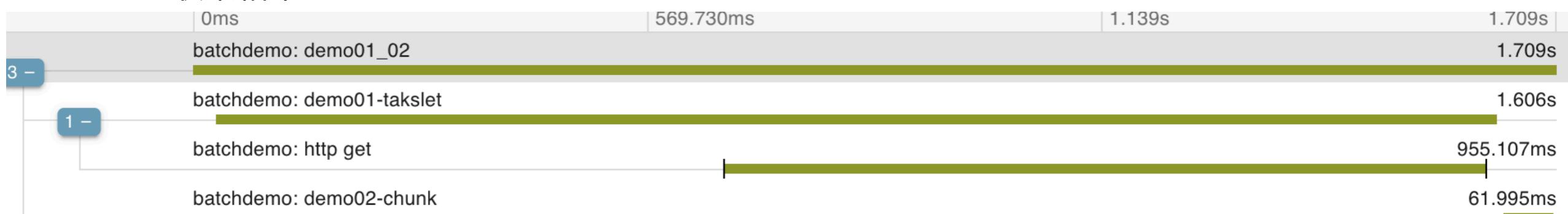
# 新機能2：Micrometerによるトレース収集

Job実行単位にトレースが切られ、Step単位にスパンが切られる

構成例



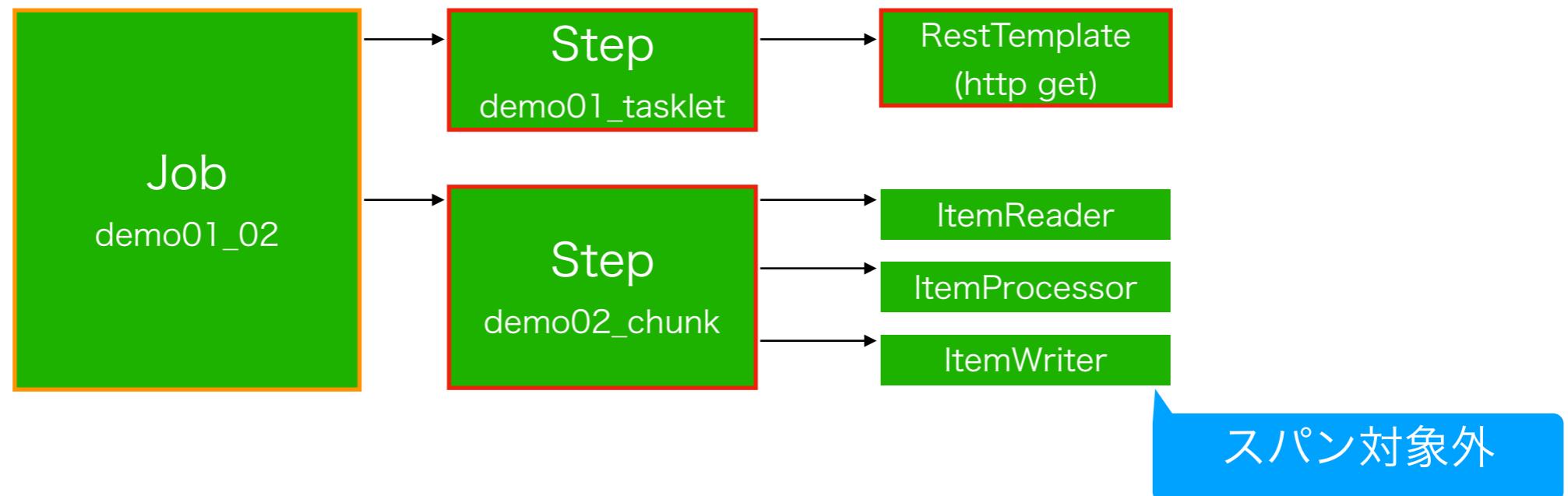
トレース収集結果



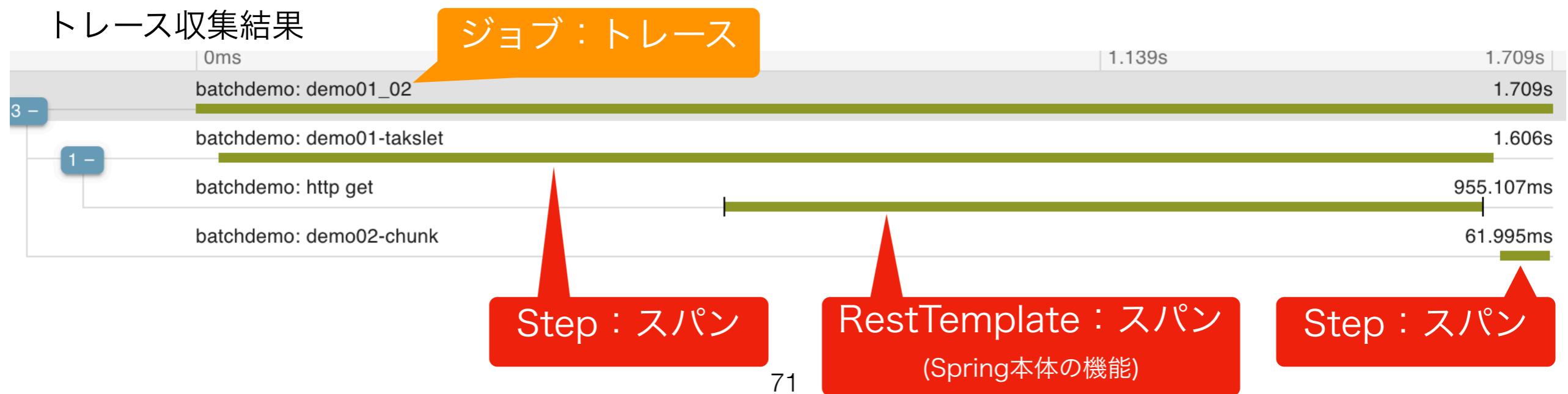
# 新機能2：Micrometerによるトレース収集

Job実行単位にトレースが切られ、Step単位にスパンが切られる

構成例



トレース収集結果



# Micrometerによるトレース収集：実装方法(Zipkin)

## Step1：以下のBeanPostProcessorをBean定義

```
@Bean  
public static BatchObservabilityBeanPostProcessor  
    batchObservabilityBeanPostProcessor() {  
  
    return new BatchObservabilityBeanPostProcessor();  
}
```

本来、`@EnableBatchProcessing`で有効になるが、  
Spring Boot利用時は`@EnableBatchProcessing`を利用できないため、  
Spring Boot 3.0.4時点では自前定義する必要がある。

<https://docs.spring.io/spring-batch/docs/5.0.0/reference/html/monitoring-and-metrics.html#tracing>

<https://github.com/spring-projects/spring-boot/pull/34305>

# Micrometerによるトレース収集：実装方法(Zipkin)

## Step2：Spring ActuatorやMicrometerの組み込み（Mavenの場合）

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing-brave</artifactId>
</dependency>
<dependency>
    <groupId>io.zipkin.reporter2</groupId>
    <artifactId>zipkin-reporter-brave</artifactId>
</dependency>
<dependency> <!-- Spring MVC/WebFlux未使用時に必要 -->
    <groupId>io.zipkin.reporter2</groupId>
    <artifactId>zipkin-sender-urlconnection</artifactId>
</dependency>
```

Brave経由でZipkinに送信する場合の例。その他のトレーサー実装利用も可能（リファレンス参照）。

<https://docs.spring.io/spring-boot/docs/3.0.4/reference/htmlsingle/#actuator.micrometer-tracing.getting-started>

# Micrometerによるトレース収集：実装方法(Zipkin)

## Step3：application.properties(or yml)にトレース関連の設定

```
spring.application.name=batchdemo  
management.tracing.sampling.probability=1.0  
management.zipkin.tracing.endpoint=http://localhost:9411/api/v2/spans
```

…probabilityはトレース収集のサンプリング率、1.0で100%収集。

…endpointはZipkinのエンドポイントを指定するが、上記はデフォルト値のため省略可能

## Step4：アプリケーションが処理完了後にすぐプロセス終了しないようにする

```
public static void main(String[] args) throws InterruptedException {  
    ConfigurableApplicationContext context  
        = SpringApplication.run(BatchDemo01Application.class, args);  
  
    Thread.sleep(10000);  
    System.exit(SpringApplication.exit(context));  
}
```

マシン状況によってはトレース情報が飛ばないケースが見られたので、ワークアラウンドとして10秒待機。

# Micrometerによるトレース収集：実装方法(Zipkin)

## Step5 : Zipkinサーバの構築

<https://zipkin.io/pages/quickstart.html>

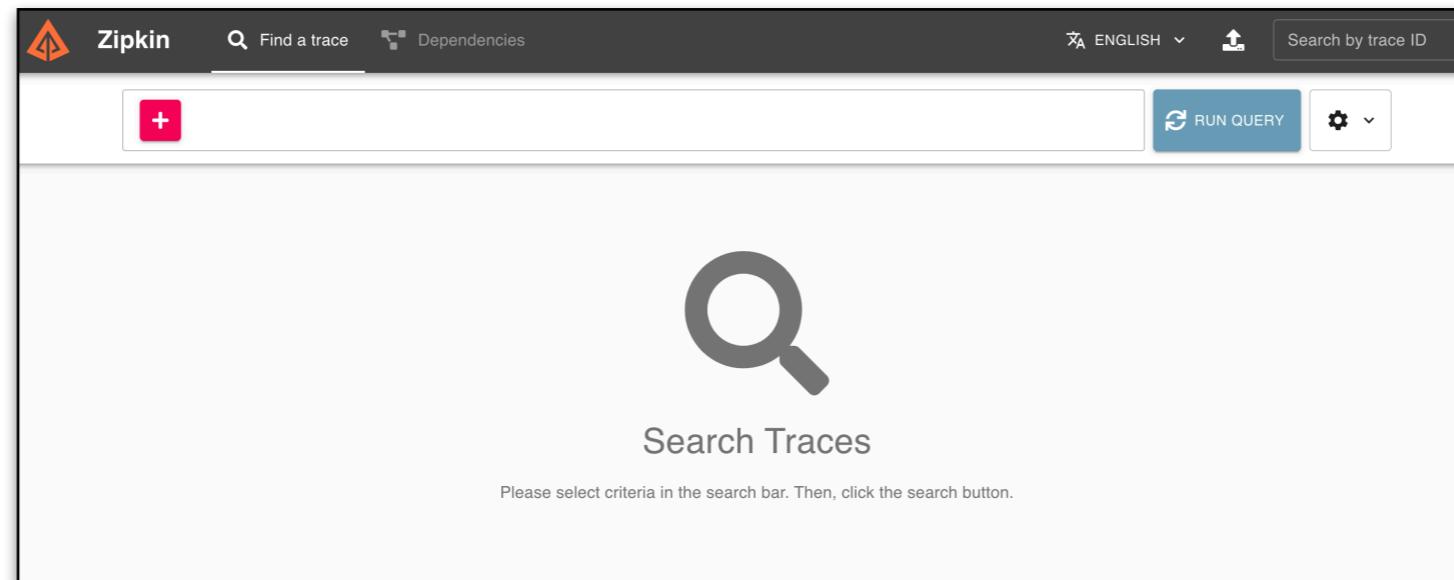
```
$ docker run -d -p 9411:9411 openzipkin/zipkin
```

Or

```
$ curl -sSL https://zipkin.io/quickstart.sh | bash -s
$ java -jar zipkin.jar
```

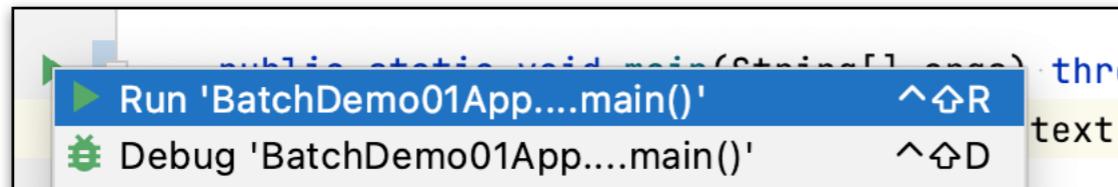
Webブラウザで右記URLにアクセス

**http://localhost:9411**



# Micrometerによるトレース収集：実装方法(Zipkin)

## Step6：バッチアプリケーションの実行



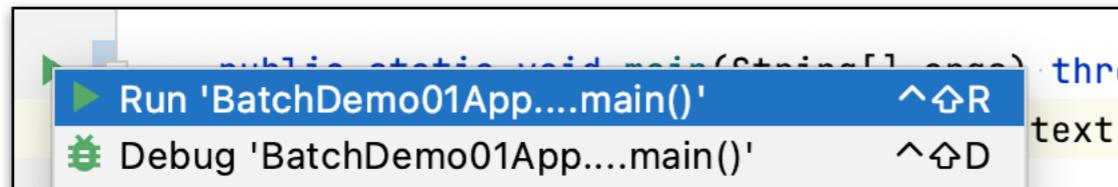
## Step7 : Zipkin画面で「RUN QUERY」を押下

The screenshot shows the Zipkin interface with a single result. The trace is identified as 'batchdemo: demo01\_02' and started 14 minutes ago. The duration of the trace is 63.424ms. The 'RUN QUERY' button is highlighted with a red box and an arrow points to it from the 'Duration' field.

トレース (=Job実行) が表示される

# Micrometerによるトレース収集：実装方法(Zipkin)

## Step6：バッチアプリケーションの実行



## Step7 : Zipkin画面で「RUN QUERY」を押下

The screenshot shows the Zipkin interface with a single trace result. The trace is identified by the service name 'batchdemo' and the trace ID 'demo01\_02'. It was started 14 minutes ago at 03/08 00:16:29:627. The trace contains 2 spans with a total duration of 63.424ms. A red box highlights the 'SHOW' button next to the duration.

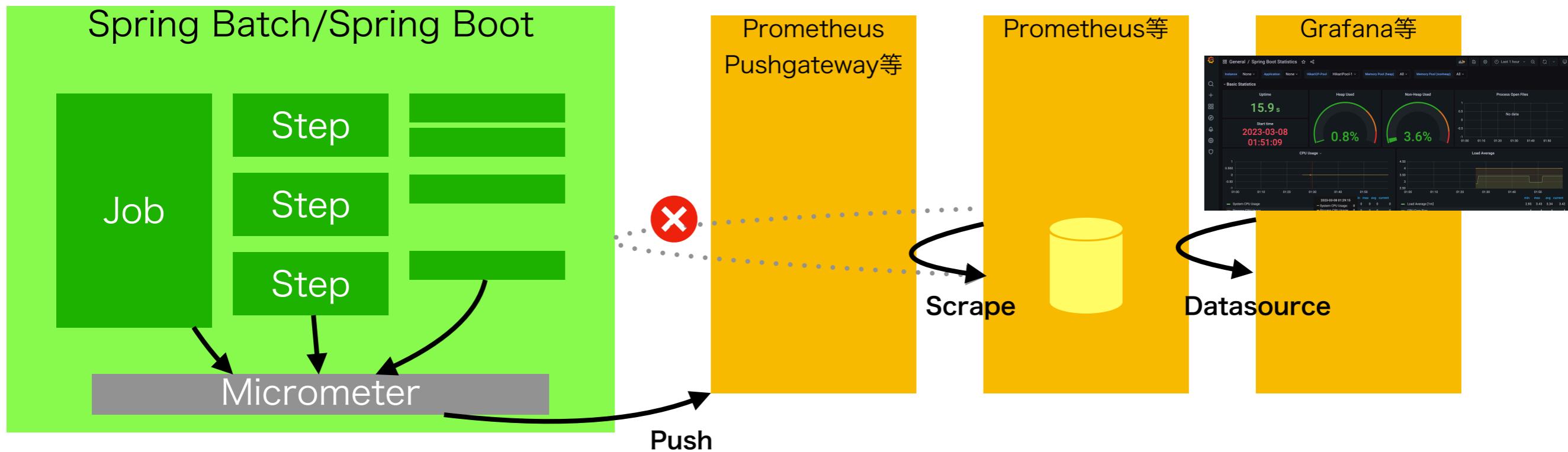
トレースに含まれるスパン (=Step実行など) が表示される



- ・新機能1：GraalVMネイティブサポート
  - ・新機能2：Micrometerによるトレース収集
  - ・新機能でないけど：Micrometerによるメトリクス収集
-

# 新機能でないけど：Micrometerによるメトリクス収集

Spring6やMicrometer1.10のObservability機能の仕組みを用いた、  
バッチ処理のメトリクス収集に対応 (Spring Batch 4.2+)



プロセスが非常駐のためPrometheusから直接pullできないため、  
pushgatewayを介してPrometheusに収集する。  
詳細は以下の楳さんの2019年のブログを参照

Cloud Foundry上で実行したSpring BatchアプリのMetricsをPrometheus Pushgatewayに送る  
<https://blog.ik.am/entries/600>

# 新機能でないけど：Micrometerによるメトリクス収集

## Spring Batchが公開するメトリクス

Metric Name	Type	Description	Tags
spring.batch.job	TIMER	Duration of job execution	name, status
spring.batch.job.active	LONG_TASK_TIMER	Currently active jobs	name
spring.batch.step	TIMER	Duration of step execution	name, job.name, status
spring.batch.step.active	LONG_TASK_TIMER	Currently active step	name
spring.batch.item.read	TIMER	Duration of item reading	job.name, step.name, status
spring.batch.item.process	TIMER	Duration of item processing	job.name, step.name, status
spring.batch.chunk.write	TIMER	Duration of chunk writing	job.name, step.name, status

<https://docs.spring.io/spring-batch/docs/5.0.0/reference/html/monitoring-and-metrics.html#built-in-metrics>

## Spring Boot標準のメトリクス

JVM情報（Heap、スレッド数等）やCPU、起動時間など

<https://docs.spring.io/spring-boot/docs/3.0.4/reference/html/actuator.html#actuator.metrics.supported>

# Micrometerによるメトリクス収集：実装方法(Pushgateway)

## Step1：Spring ActuatorやMicrometerの組み込み（Mavenの場合）

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
<dependency>
    <groupId>io.prometheus</groupId>
    <artifactId>simpleclient_pushgateway</artifactId>
</dependency>
```

<https://docs.spring.io/spring-boot/docs/3.0.4/reference/html/actuator.html#actuator.metrics.export.prometheus>

# Micrometerによるメトリクス収集：実装方法(Pushgateway)

## Step2 : application.properties(or yml)にPushgateway送信の設定

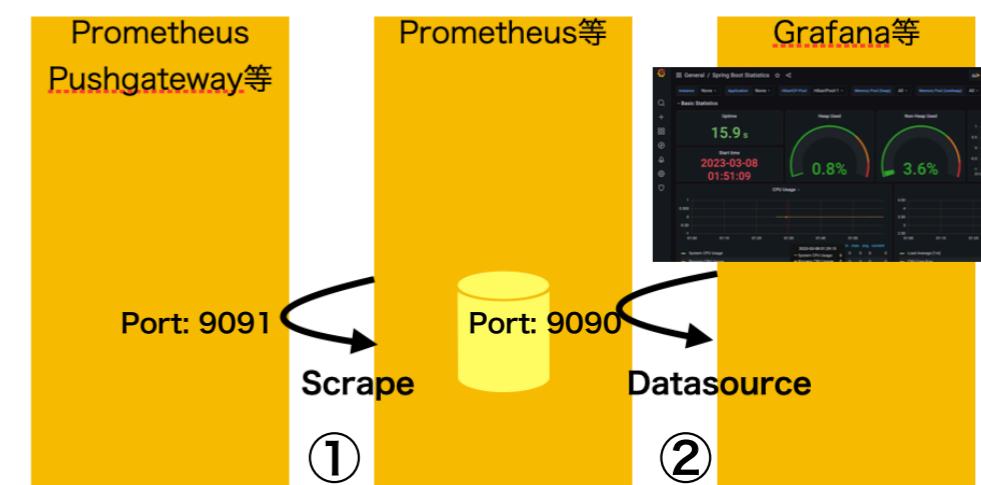
```
management.metrics.distribution.percentiles-histogram.spring.batch=true  
management.prometheus.metrics.export.pushgateway.enabled=true  
management.prometheus.metrics.export.base-url=http://localhost:9091  
management.prometheus.metrics.export.pushgateway.job=batchdemo  
management.prometheus.metrics.export.pushgateway.shutdown-operation: push
```

## Step3 : Pushgateway/Prometheus/Grafanaを構築（略）

①Prometheus→Pushgatewayのスクレイピング設定

(prometheus.yml)

```
scrape_configs:  
  - job_name: 'pushgateway'  
    honor_labels: true  
    static_configs:  
      - targets:  
        - 'host.docker.internal:9091'
```

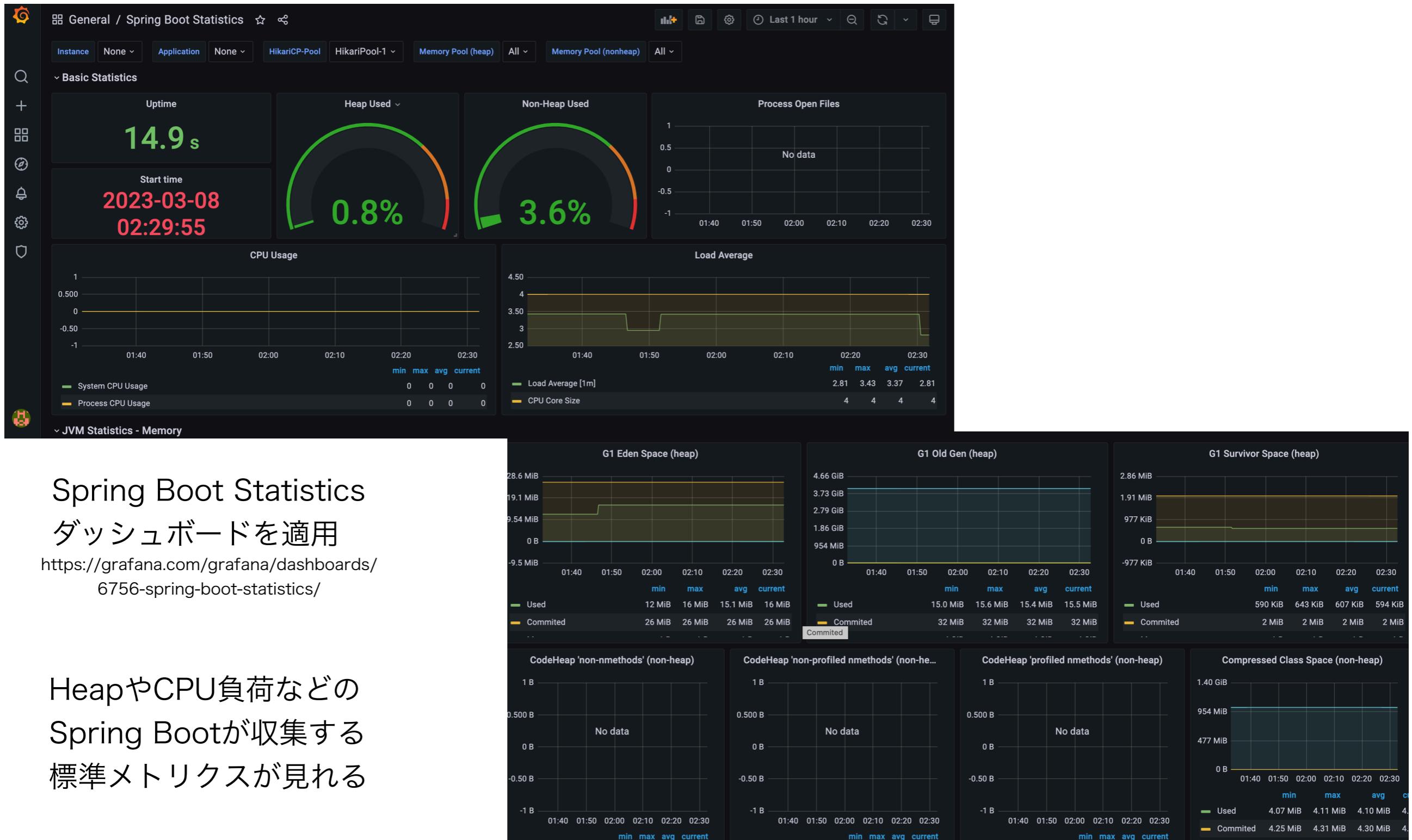


②Grafana→Prometheusのデータソース参照設定

GrafanaのGUIコンソール(Webブラウザ)から可能

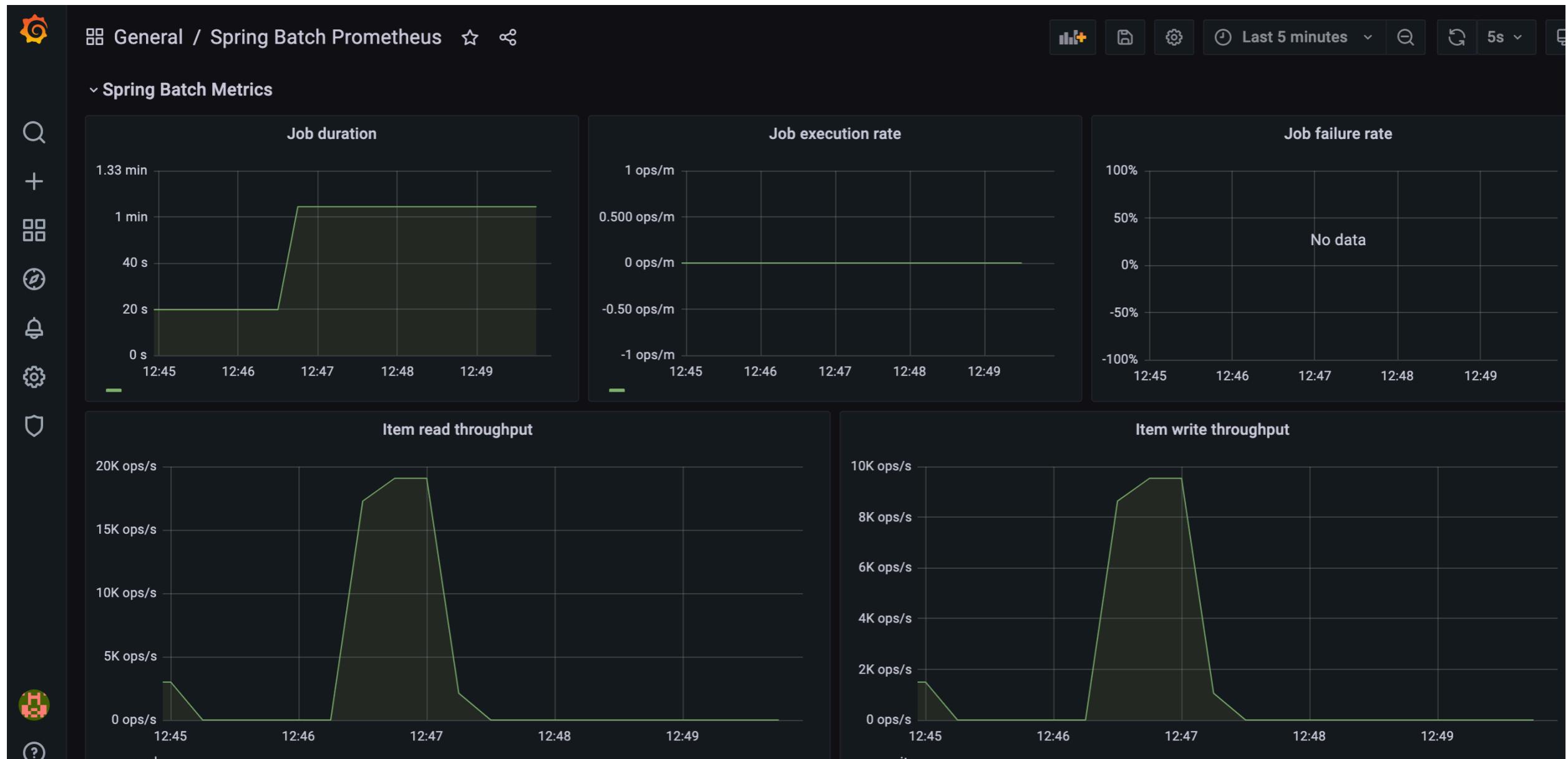
# Micrometerによるメトリクス収集：実装方法(Pushgateway)

## Step4 : Grafanaダッシュボード(Webブラウザ)での確認



# Micrometerによるメトリクス収集：実装方法(Pushgateway)

## Step5 : Grafanaダッシュボード(Webブラウザ)での確認 Spring Batchメトリクス



以下のダッシュボードを適用

<https://github.com/spring-projects/spring-batch/blob/main/spring-batch-samples/src/grafana/spring-batch-dashboard.json>

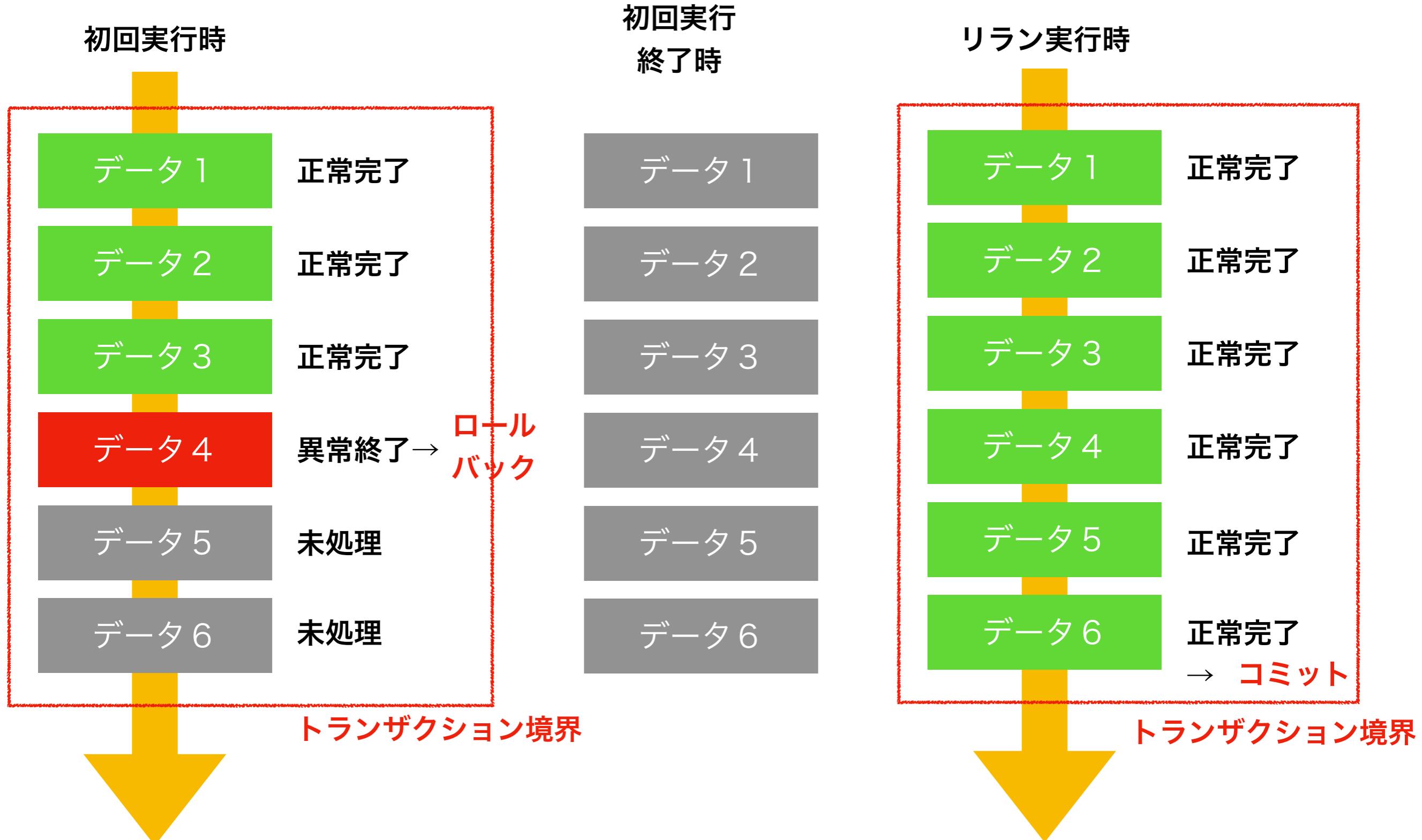
Jobの実行時間や  
チャンク処理のスループットが見える

# まとめ

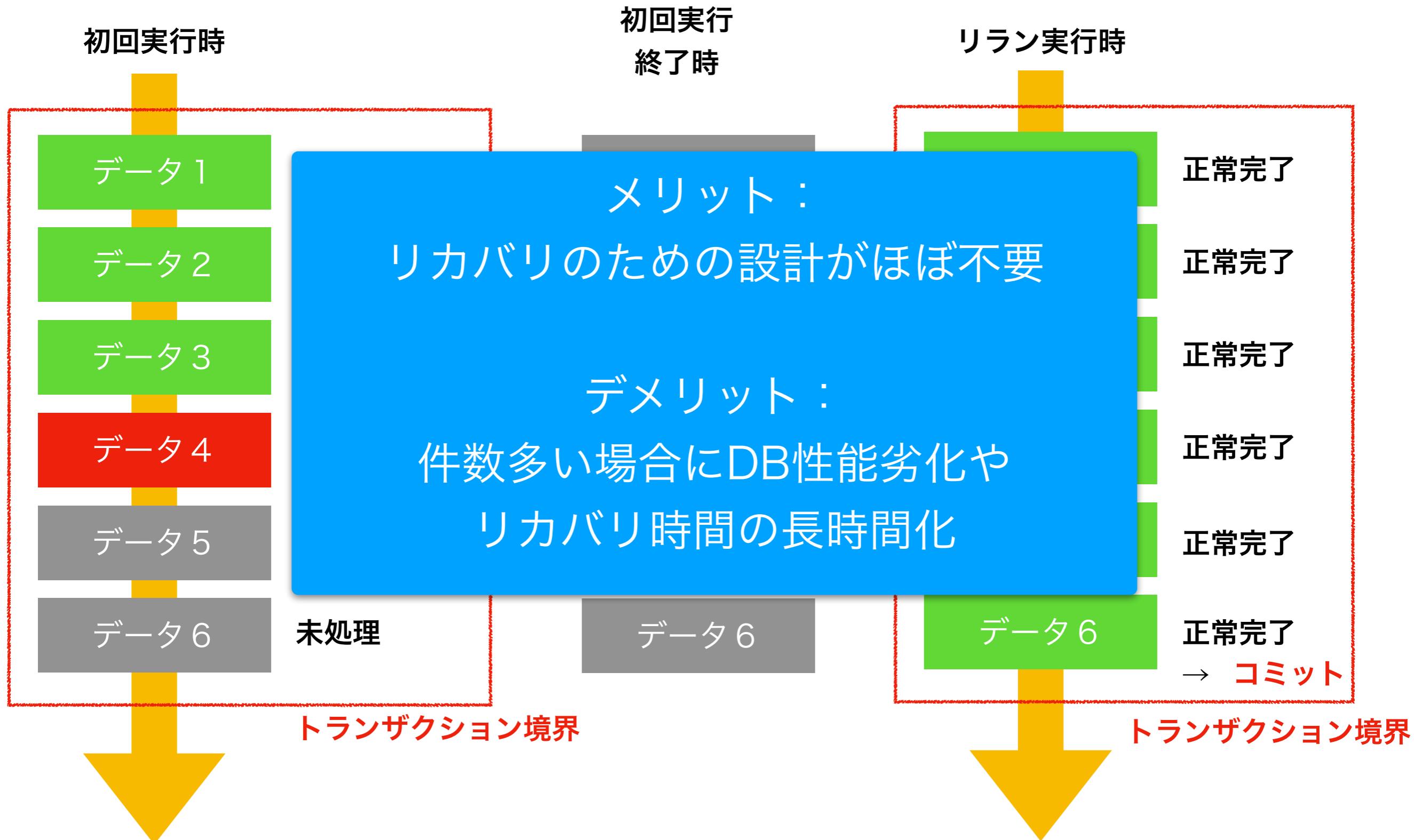
- Spring Batch5  
→Spring6世代のバッチ処理フレームワーク
- Spring Batch4からの変更点  
→Job/Step登録やDBスキーマ、  
Spring Boot利用時の@EnableXXに注意
- Spring Batch5の新機能  
→ネイティブサポート、Observability

# Appendix

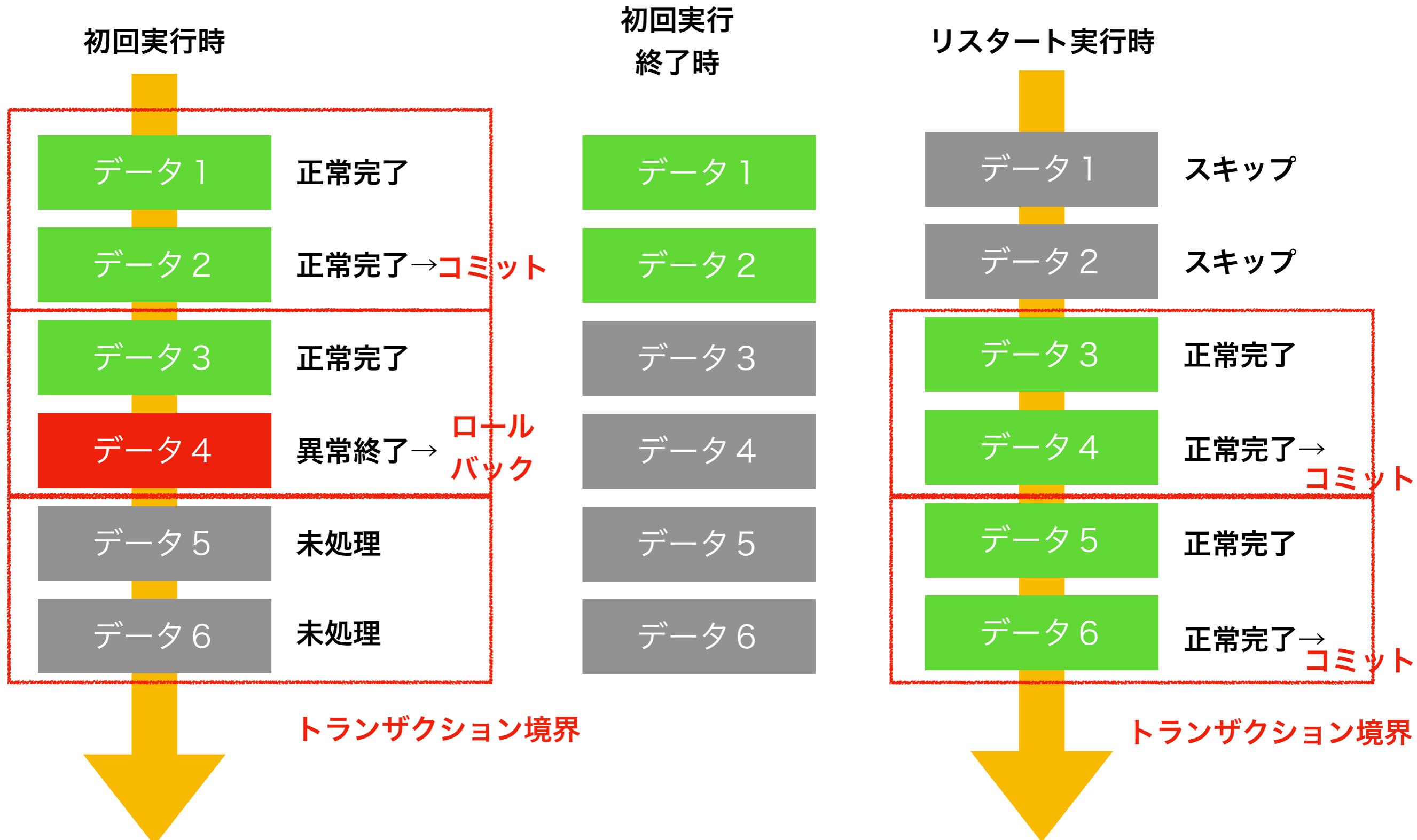
# 参考：リランによるリカバリ



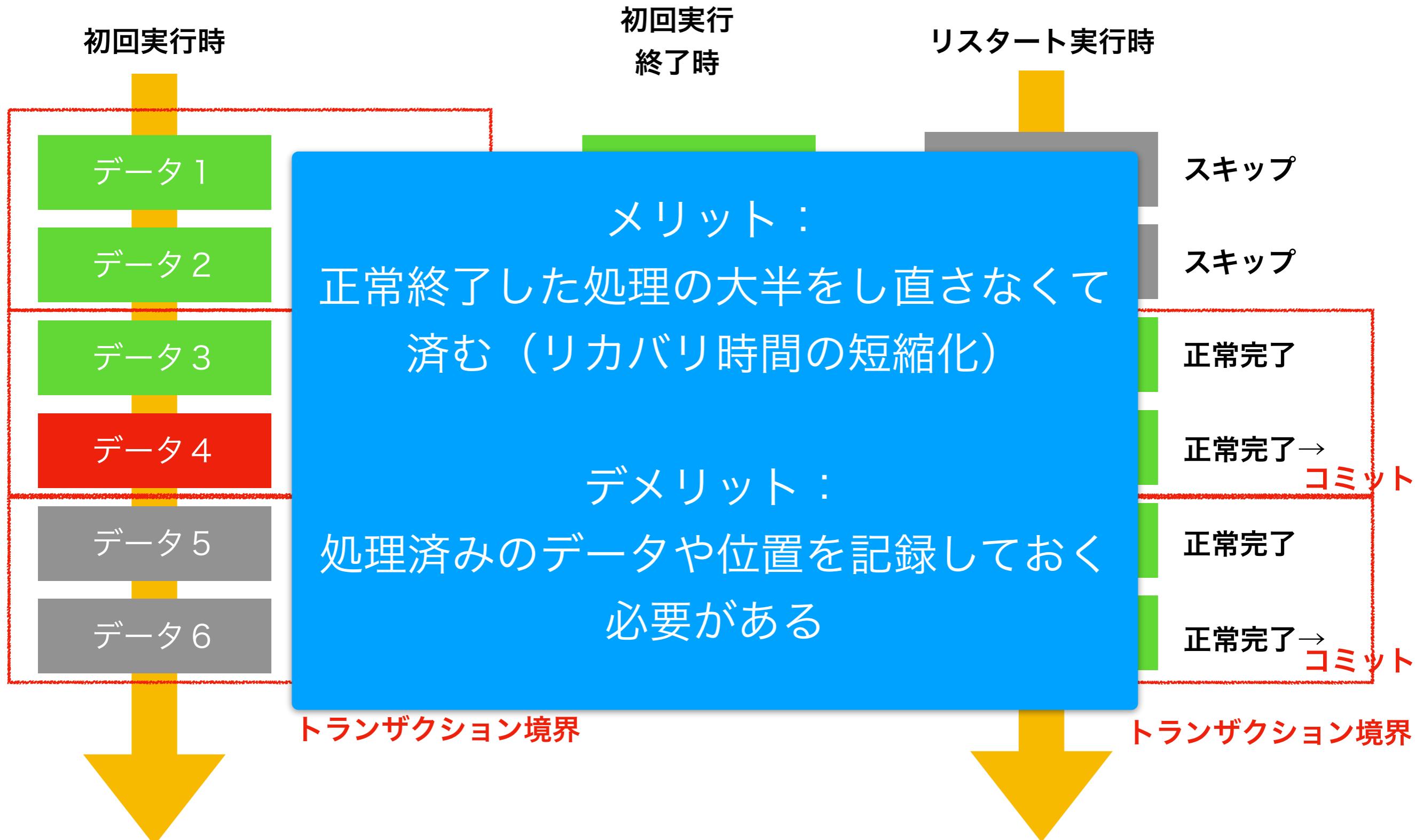
# 参考：リランによるリカバリ



# 参考：リストアによるリカバリ



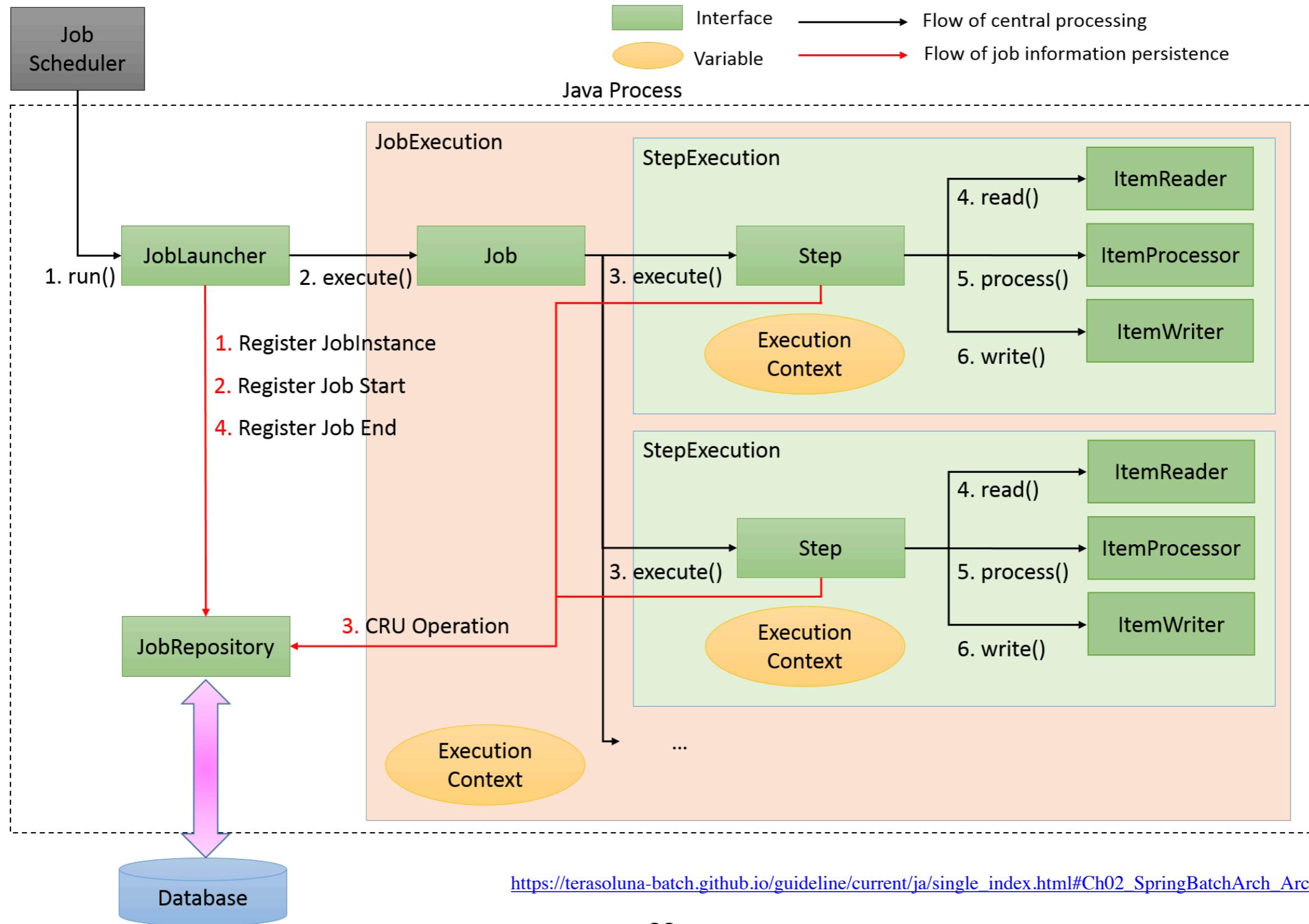
# 参考：リストアによるリカバリ



# JobRepository に関する疑問や注意事項

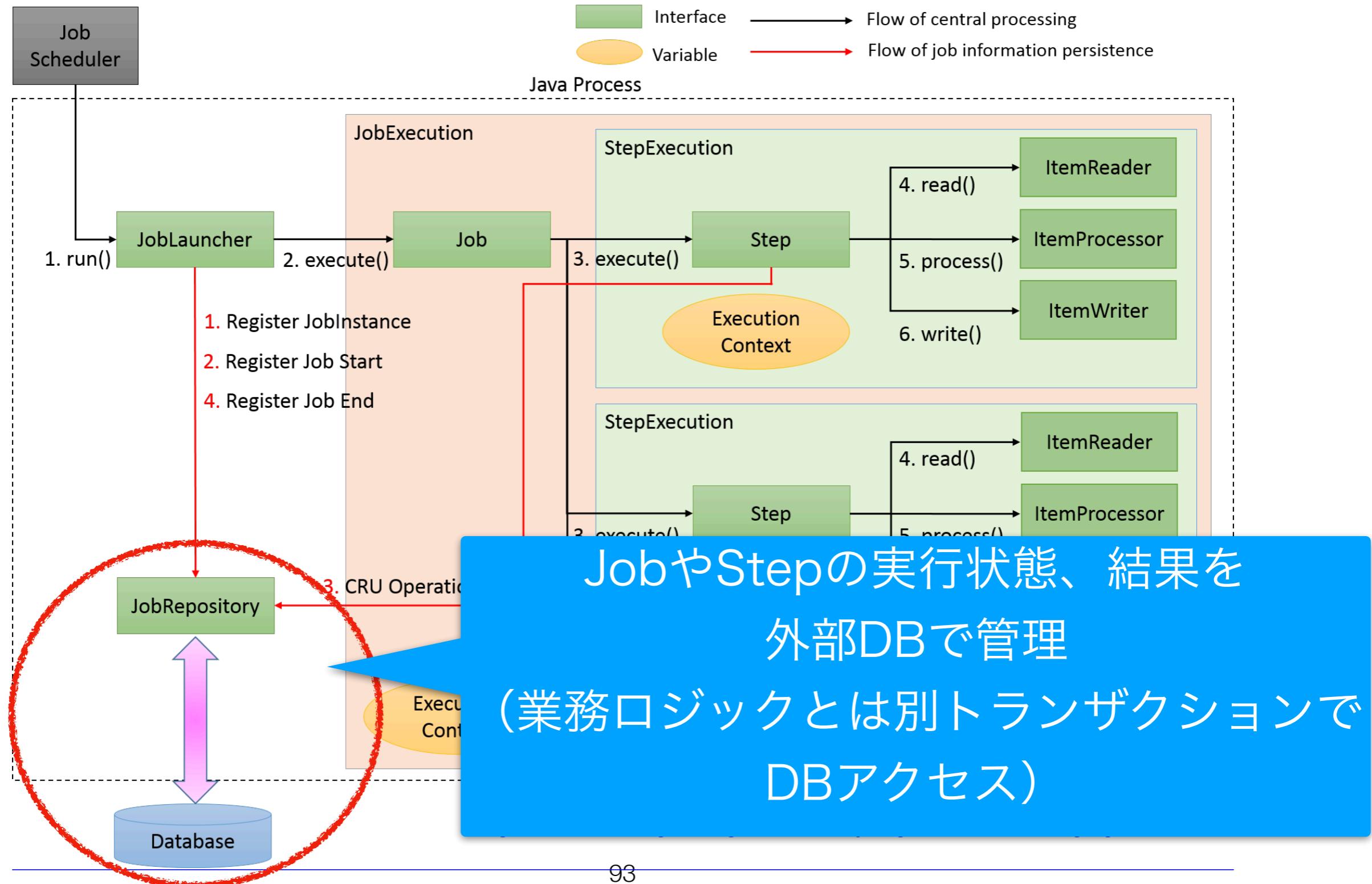
# JobRepositoryとは？

## Spring Batchの全体の流れ



# JobRepositoryとは？

## Spring Batchの全体の流れ



# JobRepositoryのテーブル構造

Job内のStepの状態・実行結果  
(Step実行の度に増加)

BATCH_STEP_EXECUTION	
○	STEP_EXECUTION_ID : BIGINT[20]
○	VERSION : BIGINT[20]
○	STEP_NAME : VARCHAR[100]
□	JOB_EXECUTION_ID : BIGINT[20]
	START_TIME : DATETIME
	END_TIME : DATETIME
	STATUS : VARCHAR[10]
	COMMIT_COUNT : BIGINT[20]
	READ_COUNT : BIGINT[20]
	FILTER_COUNT : BIGINT[20]
	WRITE_COUNT : BIGINT[20]
	READ_SKIP_COUNT : BIGINT[20]
	WRITE_SKIP_COUNT : BIGINT[20]
	PROCESS_COUNT : BIGINT[20]
	ROLLBACK_COUNT : BIGINT[20]
	EXIT_CODE : VARCHAR[100]
	EXIT_MESSAGE : VARCHAR[2500]
	LAST_UPDATE : DATETIME

BATCH_JOB_INSTANCE	
○	JOB_INSTANCE_ID : BIGINT[20]
○	VERSION : BIGINT[20]
○	JOB_NAME : VARCHAR[100]
○	JOB_KEY : VARCHAR[32]

Job (Job名×Job引数の単位で  
増加)

BATCH_STEP_EXECUTION_CONTEXT	
□	STEP_EXECUTION_ID : BIGINT[20]
	SHORT_CONTEXT : VARCHAR[2500]
	SERIALIZE_CONTEXT : TEXT

Stepコンテキストオブジェクトの  
シリアルライズデータ

BATCH_JOB_EXECUTION	
○	JOB_EXECUTION_ID : BIGINT[20]
○	VERSION : BIGINT[20]
□	JOB_INSTANCE_ID : BIGINT[20]
	CREATE_TIME : DATETIME
	START_TIME : DATETIME
	END_TIME : DATETIME
	STATUS : VARCHAR[10]
	EXIT_CODE : VARCHAR[100]
	EXIT_MESSAGE : VARCHAR[2500]
	LAST_UPDATE : DATETIME

Jobの状態・実行結果  
(Job実行の度に増加)

BATCH_JOB_EXECUTION_PARAMS	
□	JOB_EXECUTION_ID : BIGINT[20]
	TYPE_CD : VARCHAR[6]
	KEY_NAME : VARCHAR[100]
	STRING_VAL : VARCHAR[100]
	DATE_VAL : DATETIME
	LONG_VAL : BIGINT[20]
	DOUBLE_VAL : DOUBLE
	IDENTIFYING : CHAR[1]

Job引数

BATCH_JOB_EXECUTION_CONTEXT	
□	JOB_EXECUTION_ID : BIGINT[20]
	SHORT_CONTEXT : VARCHAR[2500]
	SERIALIZE_CONTEXT : TEXT

Jobコンテキストオブジェクト  
のシリアルライズデータ

# JobRepositoryのテーブル構造

Job内のStepの状態・実行結果  
(Step実行の度に増加)

BATCH_JOB_INSTANCE	
●	JOB_INSTANCE_ID : BIGINT[20]
●	VERSION : BIGINT[20]
●	JOB_NAME : VARCHAR[100]
●	JOB_KEY : VARCHAR[32]

Job (Job名×Job引数の単位で  
增加)

Jobの状態・実行結果  
(増加)

## 活用目的

実行結果、障害解析のための記録  
リストート機能向けの途中状態の維持

◆

BATCH_STEP_EXECUTION_CONTEXT	
□	STEP_EXECUTION_ID : BIGINT[20]
□	SHORT_CONTEXT : VARCHAR[2500]
□	SERIALIZE_CONTEXT : TEXT

Stepコンテキストオブジェクトの  
シリアルライズデータ

◆

BATCH_JOB_EXECUTION_CONTEXT	
□	JOB_EXECUTION_ID : BIGINT[20]
□	SHORT_CONTEXT : VARCHAR[2500]
□	SERIALIZE_CONTEXT : TEXT

Jobコンテキストオブジェクト  
のシリアルライズデータ

# JobRepositoryテーブルデータの出力例

batch\_job\_instanceテーブル (列を抜粋)

job_instance_id	job_name	job_key
1	demo01_01	d41d8cd98f00b204e9800998ecf8427e
3	demo01_01	399aaaf8689acbf6a6ed21f7a137a4856
15	demo01_01	778538c8d55ed78c127391d38c1abad8
25	demo01_02	ad6211d2dc9567640

Job名とJob引数の  
ハッシュ値

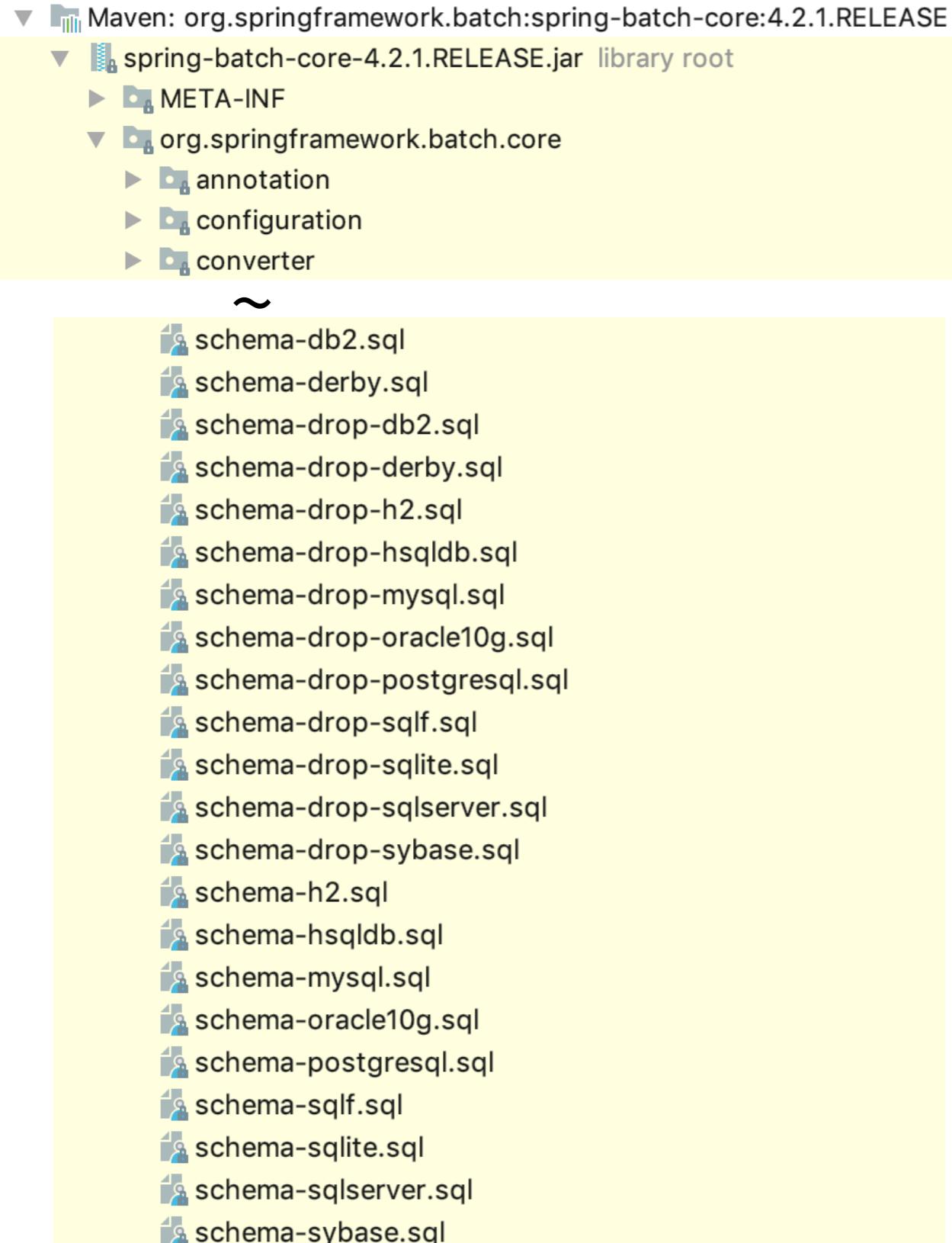
batch\_job\_executionテーブル (列を抜粋)

job_execution_id	job_instance_id	exit_code	exit_message
1	1	COMPLETED	
8	3	COMPLETED	
16	15	FAILED	java.lang.RuntimeException ...
25	25	COMPLETED	

batch\_step\_executionテーブル (列を抜粋)

step_execution_id	step_name	job_execution_id	commit_count	read_count	filter_count	write_count	exit_code	exit_message
1	demo01Takslet	1	1	0	0	0	COMPLETED	
2	demo01Takslet	8	1	0	0	0	COMPLETED	
9	demo01Takslet	16	0	0	0	0	FAILED	java.lang.RuntimeException ...
18	demo02Takslet	25	3	5	0	5	COMPLETED	

# Spring Batchが用意するDDL



## Spring BatchのJarファイル内のDDL

```
-- Autogenerated: do not edit this file

CREATE TABLE BATCH_JOB_INSTANCE (
    JOB_INSTANCE_ID BIGINT NOT NULL PRIMARY KEY ,
    VERSION BIGINT ,
    JOB_NAME VARCHAR(100) NOT NULL,
    JOB_KEY VARCHAR(32) NOT NULL,
    constraint JOB_INST_UN unique (JOB_NAME, JOB_KEY)
) ;

CREATE TABLE BATCH_JOB_EXECUTION (
    JOB_EXECUTION_ID BIGINT NOT NULL PRIMARY KEY ,
    VERSION BIGINT ,
    JOB_INSTANCE_ID BIGINT NOT NULL,
    CREATE_TIME TIMESTAMP NOT NULL,
    START_TIME TIMESTAMP DEFAULT NULL ,
    END_TIME TIMESTAMP DEFAULT NULL ,
    STATUS VARCHAR(10) ,
    EXIT_CODE VARCHAR(2500) ,
    EXIT_MESSAGE VARCHAR(2500) ,
    LAST_UPDATED TIMESTAMP,
    JOB_CONFIGURATION_LOCATION VARCHAR(2500) NULL,
    constraint JOB_INST_EXEC_FK foreign key (JOB_INSTANCE_ID)
        references BATCH_JOB_INSTANCE(JOB_INSTANCE_ID)
) ;

...
```

# ジョブ起動時にJobRepositoryテーブルを生成

Springの機能で実施

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

Spring BatchのJarファイル内のDDLの場所

Spring Bootの機能で実施

→ 組込DBを利用している場合のみ、それを自動で検出しテーブルを生成

```
spring.batch.initialize-schema=always #強制的にテーブル生成をONしたい場合（省略可）  
spring.batch.initialize-schema=never #強制的にテーブル生成をOFFしたい場合（省略可）
```

注意：

組込DBを使うような、ローカル環境やCI/CD環境での利用が一般的。  
本番環境等、DBが管理されている環境では非推奨。

# JobRepositoryを永続化する場合の注意

- レコードを定期的に退避 or 削除
  - バッチの実行毎にレコードが増加していくため
  - 削除時は外部キー制約に注意
- マルチバイト文字を考慮したレコード長 (OracleDB等)
  - VARCHAR2(2500) -> VARCHAR2(**2500 CHAR**)
  - さもないと、レコード長超えて書き込みし、異常終了
- 性能劣化防止のため各TBLにインデックスを付けること
  - Spring Batch標準DDLはインデックスを作らない
  - インデックス推奨対象はリファレンスに指南あり

<https://docs.spring.io/spring-batch/docs/current/reference/html/index-single.html#metaDataArchiving>

# 参考：インデックス推奨カラム

Spring Batchリファレンスより

**Table 21. Where clauses in SQL statements (excluding primary keys) and their approximate frequency of use.**

Default Table Name	Where Clause	Frequency
BATCH_JOB_INSTANCE	JOB_NAME = ? and JOB_KEY = ?	Every time a job is launched
BATCH_JOB_EXECUTION	JOB_INSTANCE_ID = ?	Every time a job is restarted
BATCH_STEP_EXECUTION	VERSION = ?	On commit interval, a.k.a. chunk (and at start and end of step)
BATCH_STEP_EXECUTION	STEP_NAME = ? and JOB_EXECUTION_ID = ?	Before each step execution

<https://docs.spring.io/spring-batch/docs/current/reference/html/index-single.html#recommendationsForIndexingMetaTables>

# JobRepositoryを永続化する場合の注意

- 同じジョブを、前回と同じ引数で実行した場合、同じジョブの再実行（リスタート）とみなされる
  - 前回の実行が正常終了していた場合、実行が抑止
  - タイムスタンプ等の引数で都度引数を変更する必要

同じジョブを、前回と同じ引数で実行した時の実行ログ

```
: Running default command line with: []
: Job: [SimpleJob: [name=demo01]] launched with the following parameters: []
: Step already complete or not restartable, so no action to execute:
StepExecution: id=1, version=3, name=demo01 Takslet, status=COMPLETED,
...
```

ジョブ引数の一致から、  
同一Stepが実行済みと判断され、実行抑止

# 寄り道：Spring BatchのBean定義の特徴

```
@Bean  
@StepScope  
public FlatFileItemWriter fileItemWriter(  
    @Value("#{jobParameters['outputFile']}") String filePath) {  
  
    // 中略  
  
    FlatFileItemWriter<ProcessedPerson> fileItemWriter  
        = new FlatFileItemWriter<>();  
    fileItemWriter.setResource(new FileSystemResource(filePath));  
    fileItemWriter.setLineAggregator(aggregator);  
    return fileItemWriter;  
}
```

# 寄り道：Spring BatchのBean定義の特徴

## Spring Batch独自のBean Scope

Stepの実行毎に新たなBeanインスタンスを生成  
(late-binding利用のため必須)

```
@Bean  
@StepScope  
public FlatFileItemWriter fileItemWriter(  
    @Value("#{jobParameters['outputFile']}") String filePath) {  
  
    // 中略  
  
    FlatFileItemWriter<ProcessedPerson> fileItemWriter  
        = new FlatFileItemWriter<>();  
    fileItemWriter.setResource(new FileSystemResource(filePath));  
    fileItemWriter.setLineAggregator(aggregator);  
    return fileItemWriter;  
}
```

## プロパティの遅延解決 (late binding)

ジョブ引数はSpringのDIコンテナ上の部品で  
チェックや格納がされるため、コンテナ初期化時に  
は解決できず、Stepの開始時に解決する  
(通常の@Valueはコンテナ初期化時に解決)