

知っておきたい Spring Batch Tips

JSUG 勉強会 2020/5/13

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>



自己紹介

- ・日本Springユーザ会(JSUG)スタッフ

読者様へ

- ・某S
本編に収めきれなかった
（実態）

幻の15章「Spring Batch」
あります！

- ・某書の筆者の一人

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



Spring Batchの過去の勉強会との関係性

JSUG勉強会@2017/5/25

TERASOLUNA Batch Framework
(Spring Batchがベース)



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

The screenshot shows the Table of Contents for the development guideline. The contents include:

- 1.はじめに
 - 1.1.利用規約
 - 1.1.1.参考
 - 1.2.導入
 - 1.2.1.ガイドラインの目的
 - 1.2.2.ガイドラインの対象読者
 - 1.2.3.ガイドラインの構成
 - 1.2.4.ガイドラインの読み方
 - 1.2.5.ガイドラインの動作検証環境
 - 1.3.更新履歴- 2.TERASOLUNA Batch Framework for Java (5.x)のコンセプト
 - 2.1.一般的なバッチ処理
 - 2.1.1.一般的なバッチ処理とは
 - 2.1.2.バッチ処理に求められる要件
 - 2.1.3.バッチ処理で考慮する原則と注意点
 - 2.2.TERASOLUNA Batch Framework for Java (5.x)のスタック
 - 2.2.1.概要
 - 2.2.2.TERASOLUNA

TERASOLUNA Batch Framework for Java (5.x) Development Guideline - version 5.2.1.RELEASE, 2019-5-31 > INDEX

TERASOLUNA Batch Framework for Java (5.x) Development Guideline

NTT DATA Corporation. – Version 5.2.1.RELEASE, 2019-5-31

1. はじめに

1.1. 利用規約

本ドキュメントを使用するにあたり、以下の規約に同意していただく必要があります。同意いただけない場合は、本ドキュメント及びその複製物の全てを直ちに消去又は破棄してください。

1. 本ドキュメントの著作権及びその他一切の権利は、NTTデータあるいはNTTデータに権利を許諾する第三者に帰属します。
2. 本ドキュメントの一部または全部を、自らが使用する目的において、複製、翻訳、翻案することができます。ただし本ページの規約全文、およびNTTデータの著作権表示を削除することはできません。
3. 本ドキュメントの一部または全部を、自らが使用する目的において改変したり、本ドキュメントを用いた二次的著作物を作成することができます。ただし、「参考文献：TERASOLUNA Batch Framework for Java (5.x) Development Guideline」あるいは同等の表現を、作成したドキュメント及びその複製物に記載するものとします。

https://terasoluna-batch.github.io/guideline/current/ja/single_index.html

前回：Spring Batchの概要とTERASOLUNA Batchの補完機能を紹介



今回：TERASOLUNA Batchで公開している意外と重要な
Spring BatchのTips (+ α) を厳選紹介

アジェンダ

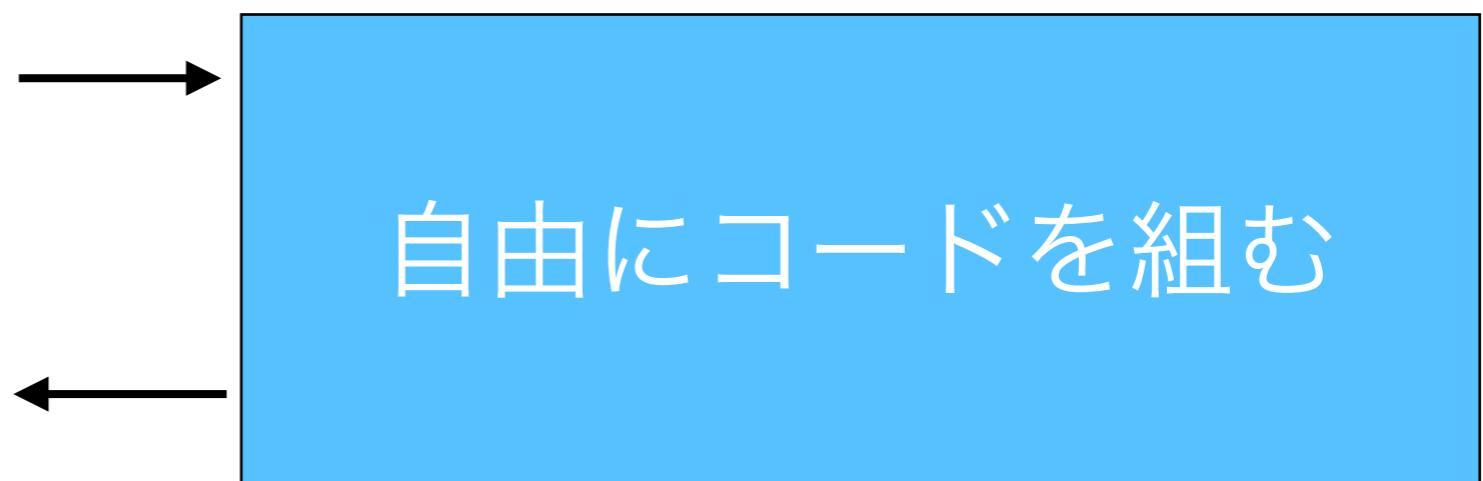
- Tasklet v.s. Chunk
- JobRepositoryに関する疑問や注意事項
- Spring BootでSpring Batchを使うべき？

Tasklet v.s. Chunk

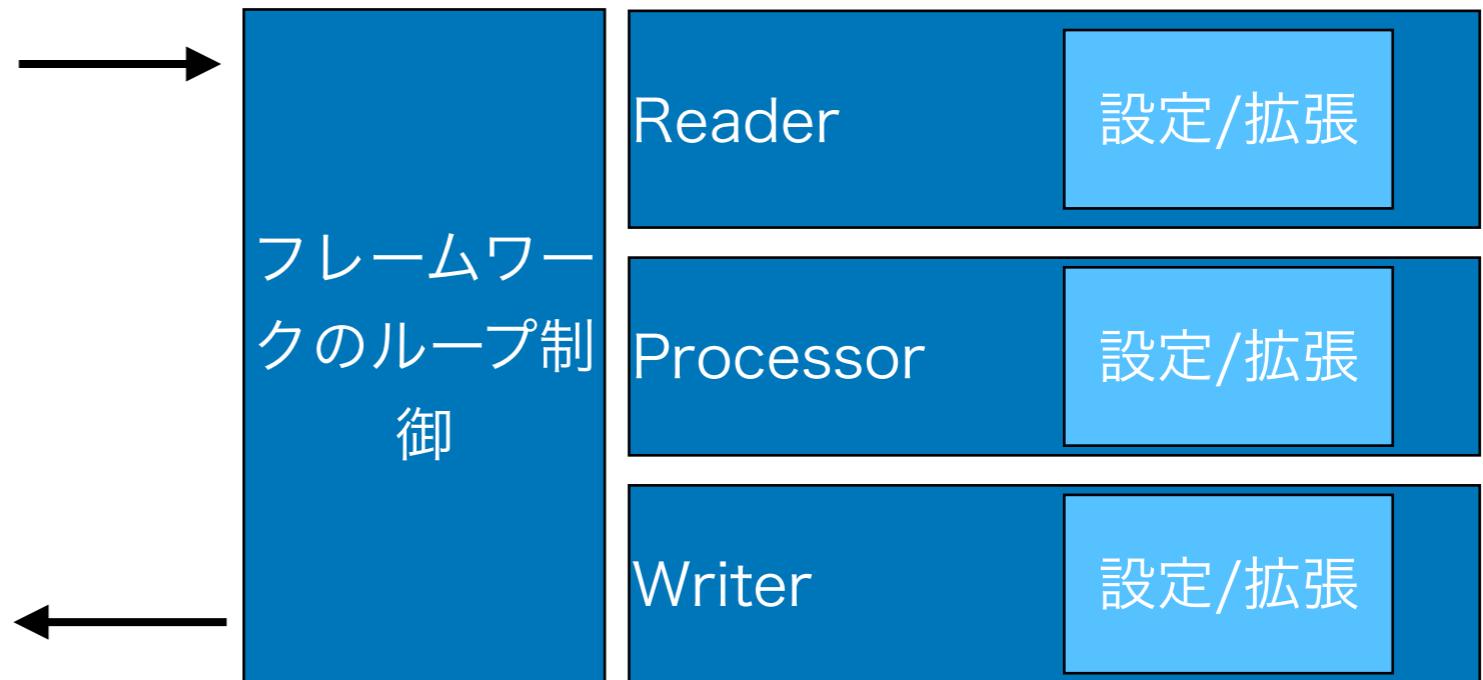
Taskletとは？Chunkとは？

ざっくりの違い

Tasklet方式
業務ロジック



Chunk方式
業務ロジック



Taskletの実装例

```
@Bean  
@StepScope  
public Tasklet demo01Takslet() {  
    return new Tasklet() {  
        @Override  
        public RepeatStatus execute(StepContribution stepContribution,  
                                    ChunkContext chunkContext) throws Exception {  
            log.info("test tasklet 01.");  
            // 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

Job引数などは、
メソッド引数より参照可能

```
@Bean  
public Step demo01Step(@Qualifier("demo01Takslet") Tasklet tasklet) {  
    return stepBuilderFactory.get("demo01Takslet")  
        .tasklet(tasklet).build();  
}
```

Tasklet BeanをStepに登録

Chunkの実装例(1/2)

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

ReaderのBean定義
(例ではCSVファイル入力)

ProcessorのBean定義

Chunkの実装例(2/2)

```
@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;  
}
```

WriterのBean定義
(例ではCSVファイル出力)

```
@Bean  
public Step demo02Step(ItemReader<OriginalPerson> reader,  
                      ItemProcessor<OriginalPerson, ProcessedPerson> processor,  
                      ItemWriter<ProcessedPerson> writer) {  
    return stepBuilderFactory.get("demo02Chunk")  
        .<OriginalPerson, ProcessedPerson> chunk(2) // チャンクサイズ設定  
        .reader(reader)  
        .processor(processor)  
        .writer(writer)  
        .build();  
}
```

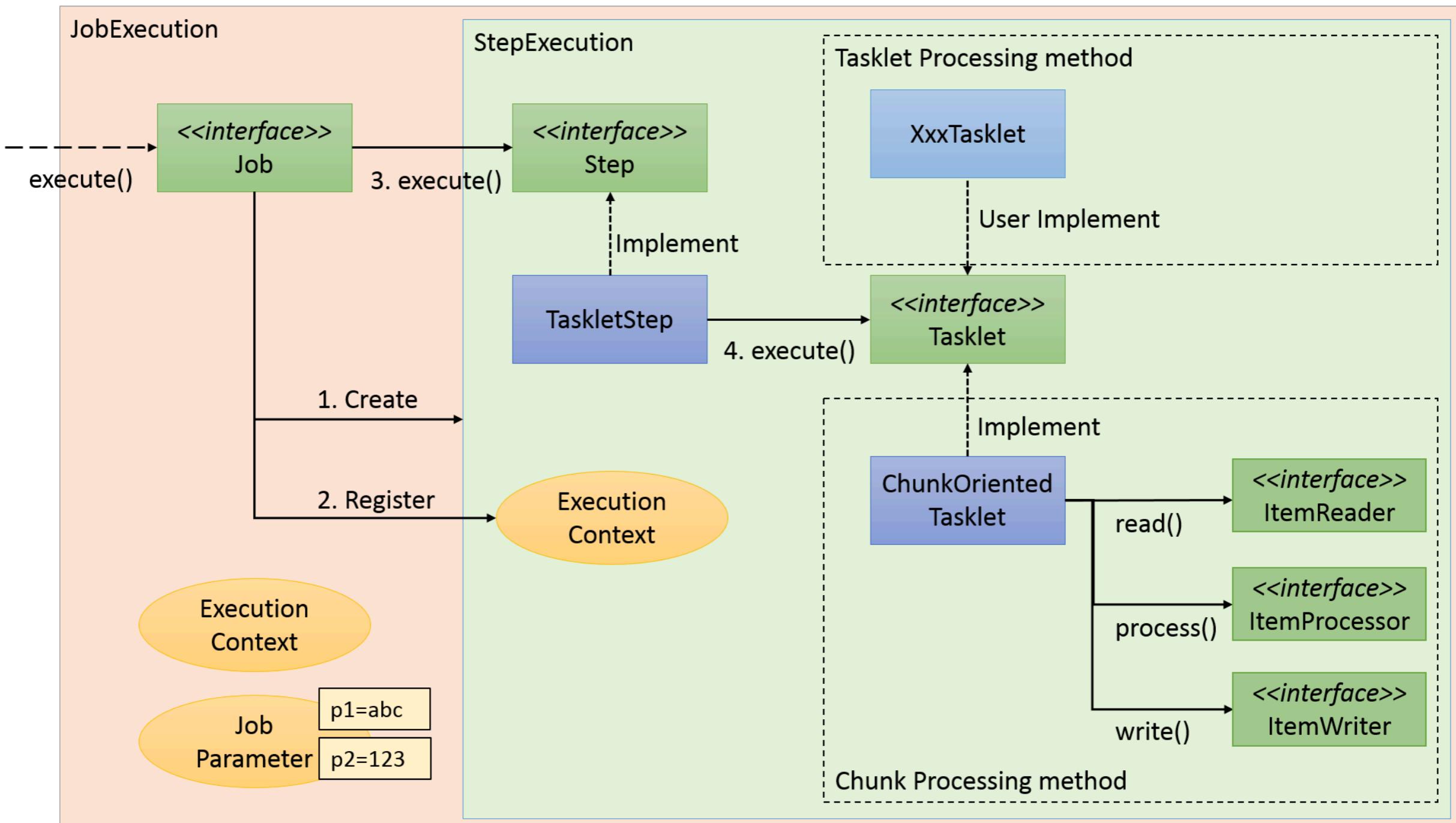
Reader/Processor/Writer
各BeanをStepに登録

参考：代表的なReader/Processor/Writer

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

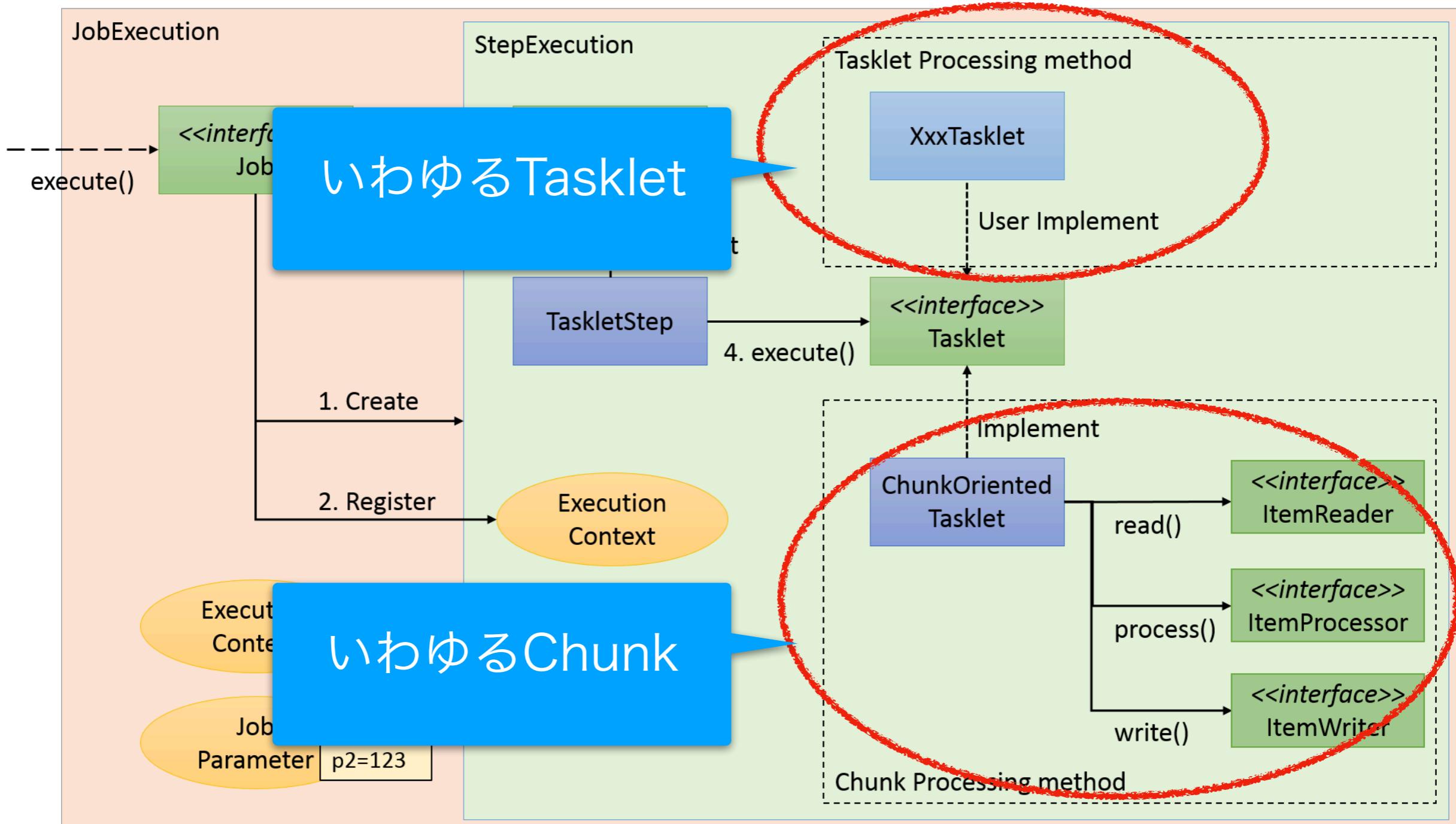
Taskletとは？Chunkとは？

Spring Batch全体像



Taskletとは？Chunkとは？

Spring Batch全体像

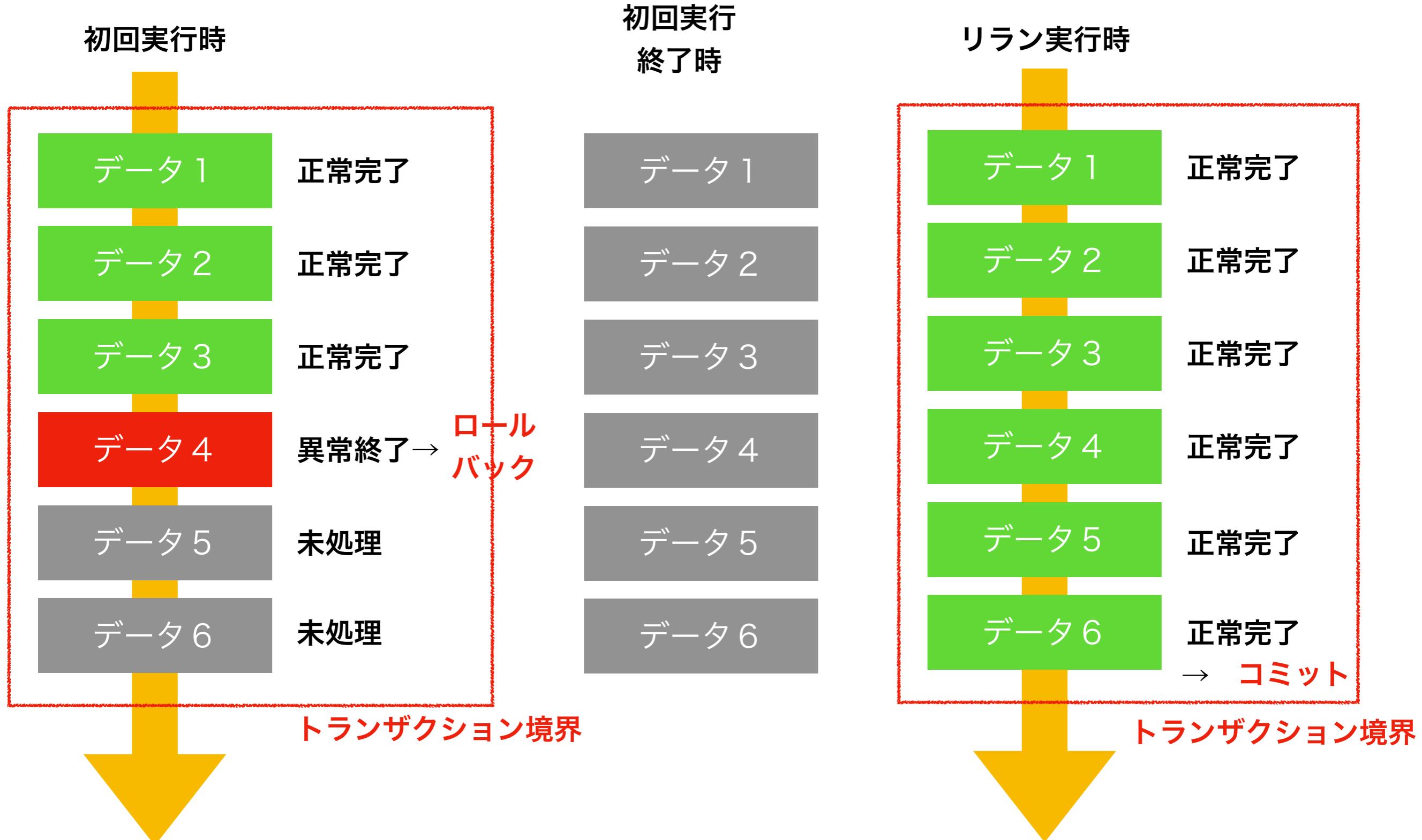


特徴

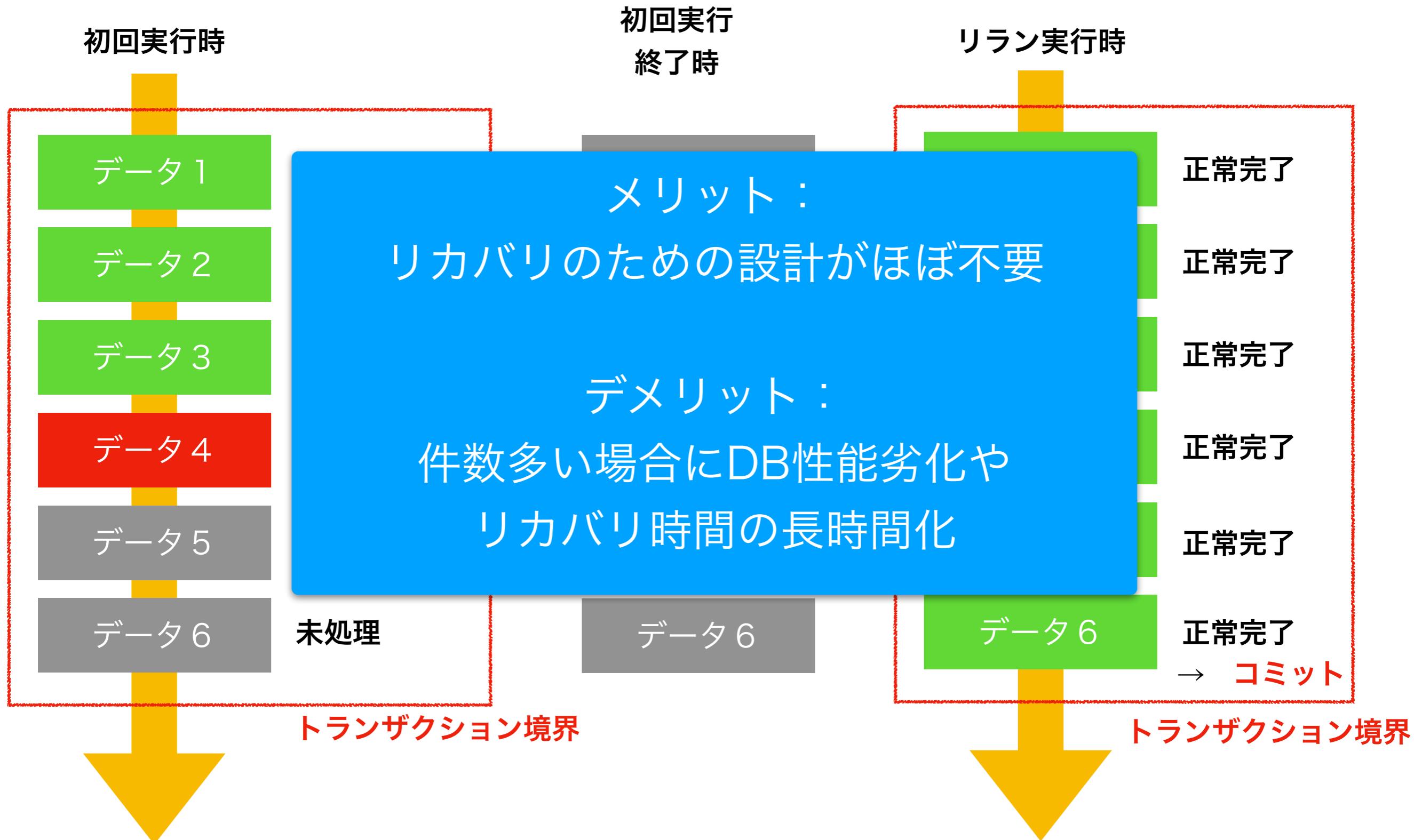
観点	Tasklet	Chunk	考察
業務ロジック記述の自由度	自由度高い (ロジック全てを開発者が記述)	自由度低い (Reader、Processor、Writerの組み合わせ)	Chunkは設計時点で構造意識が必要で、Taskletよりも学習コスト高い。
トランザクション	1トランザクション	件数／チャンクサイズに分割	リランでリカバリしたい場合はTasklet、リスタートでリカバリや大量データ処理にはChunk。
リストア	未対応	対応	

https://terasoluna-batch.github.io/guideline/current/ja/single_index.html#Ch03_ChunkOrTasklet

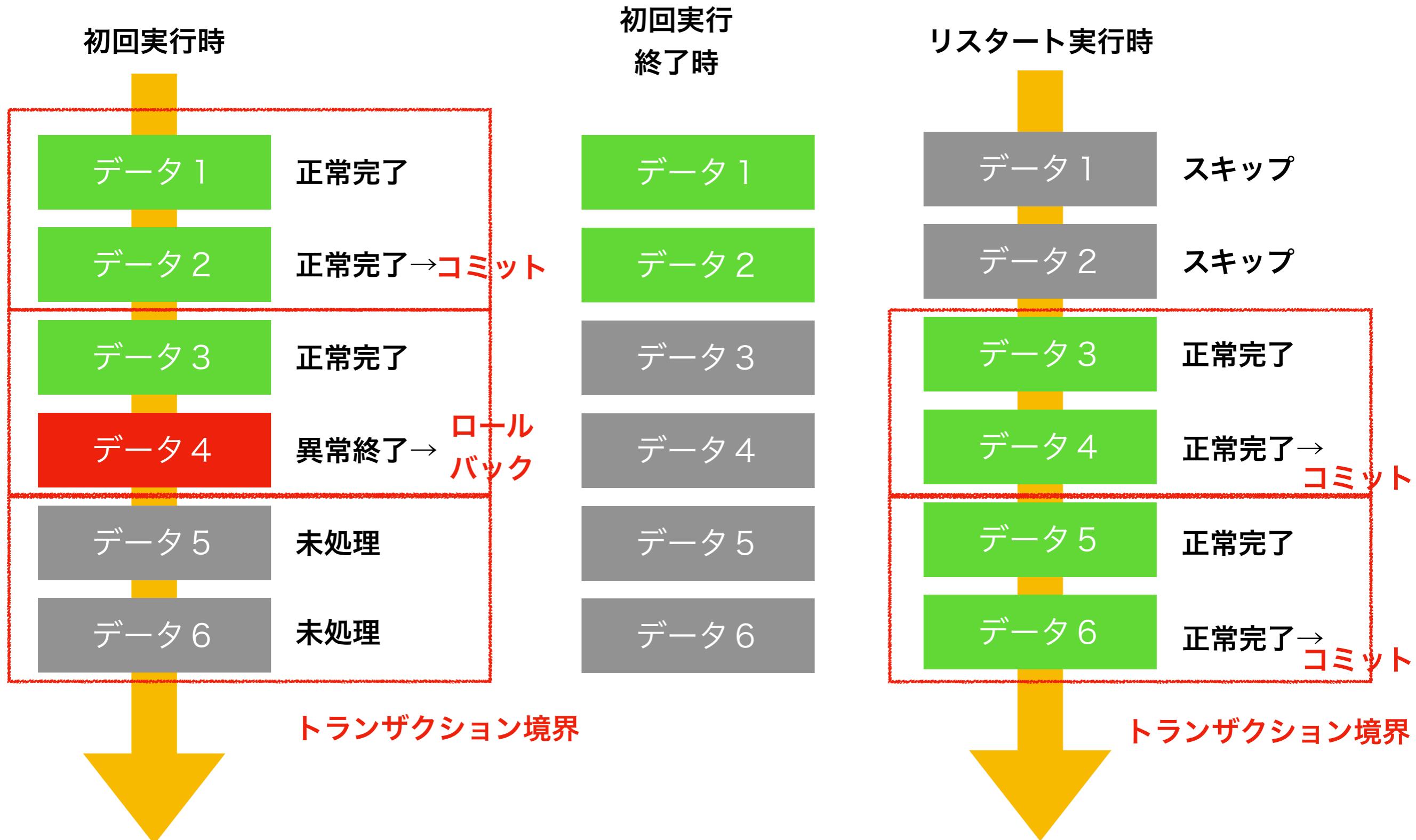
参考：リランによるリカバリ



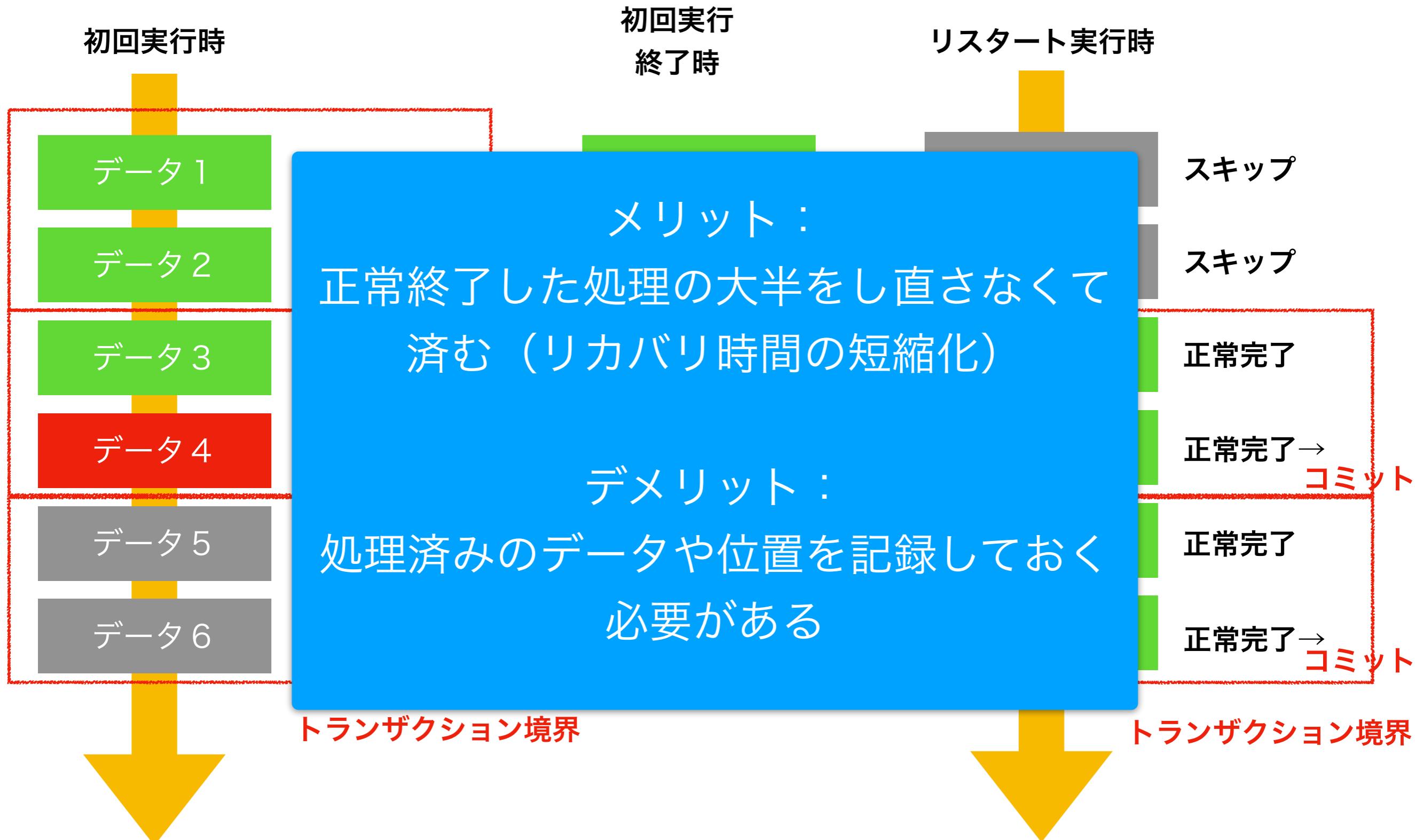
参考：リランによるリカバリ



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



特徴（再掲）

観点	Tasklet	Chunk	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

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;  
}
```

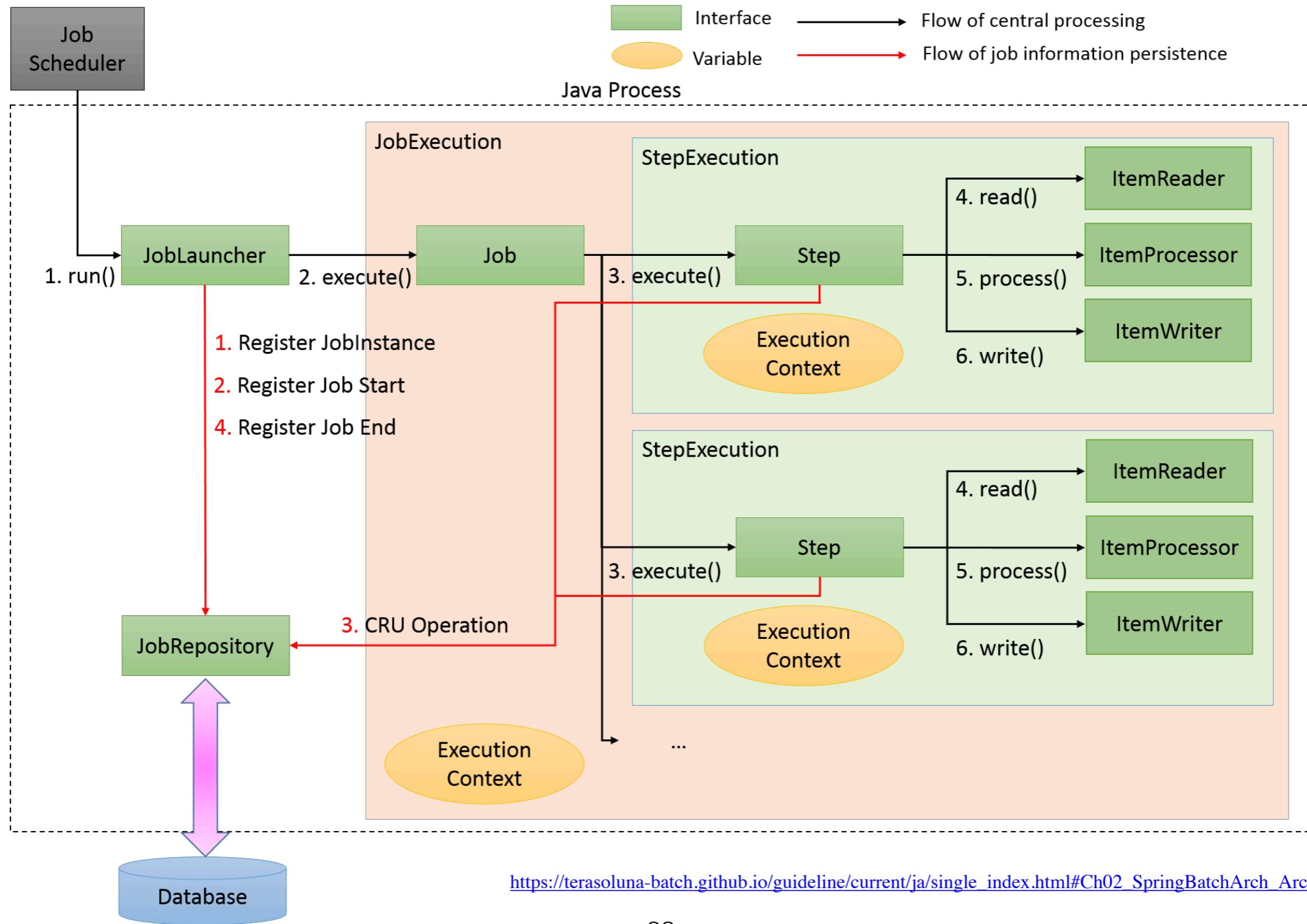
プロパティの遅延解決 (late binding)

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)

JobRepository に関する疑問や注意事項

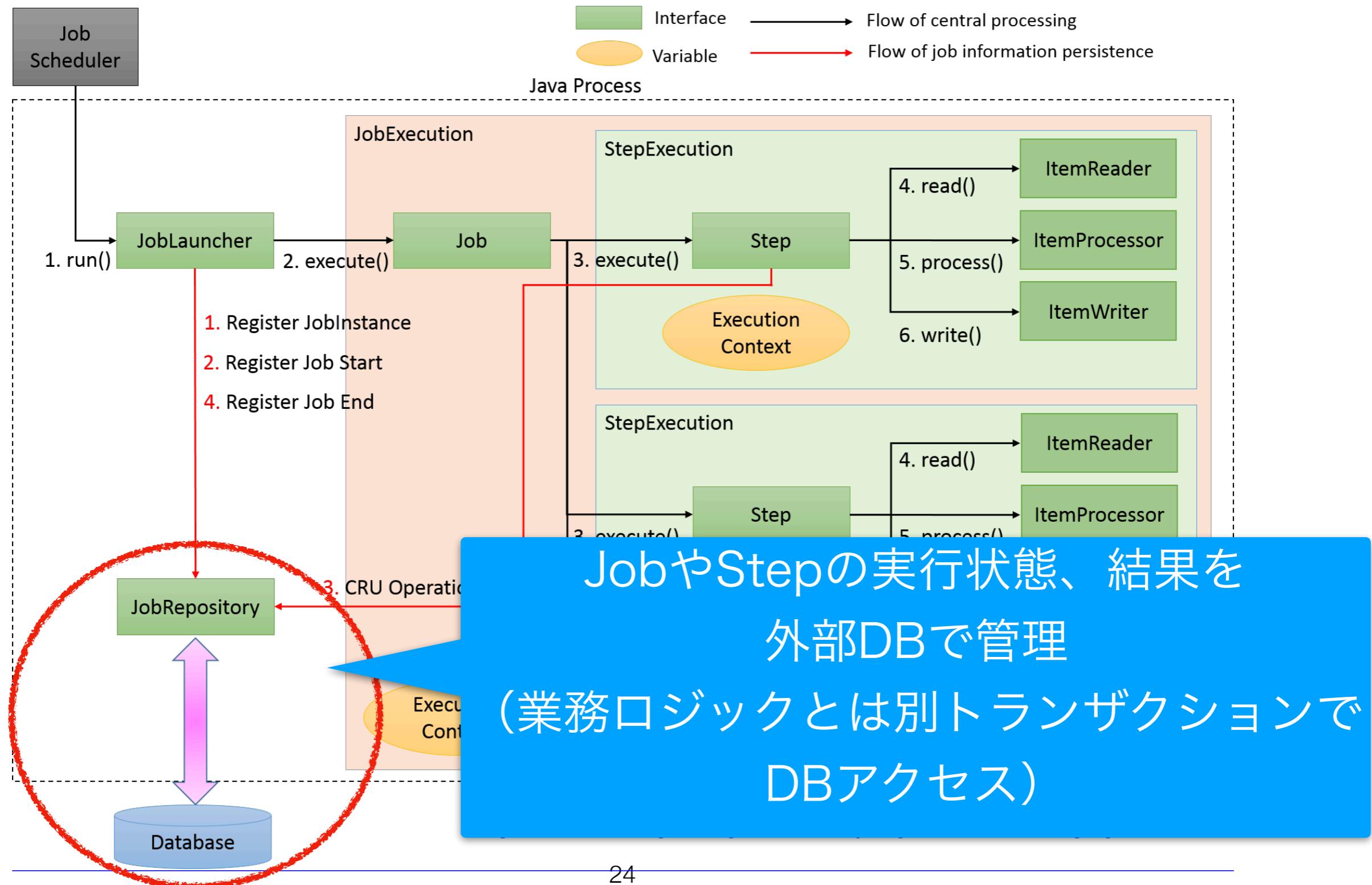
JobRepositoryとは？

Spring Batchの全体の流れ

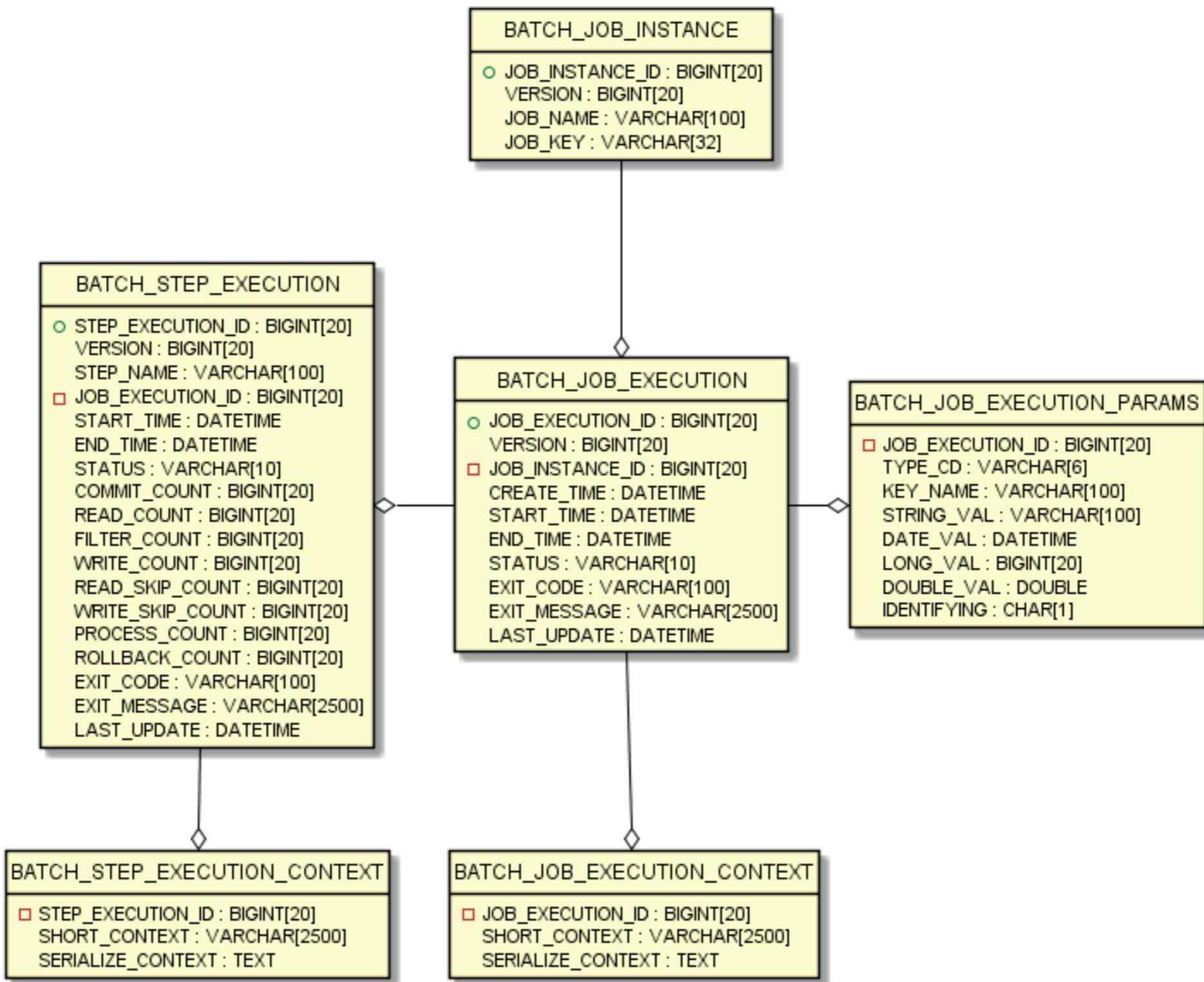


JobRepositoryとは？

Spring Batchの全体の流れ



JobRepositoryのテーブル構造



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]

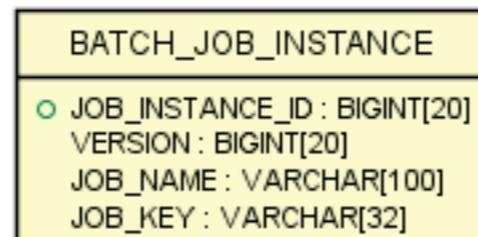
Job引数

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)

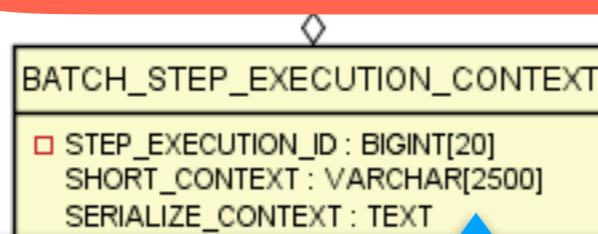


Job (Job名×Job引数の単位で
增加)

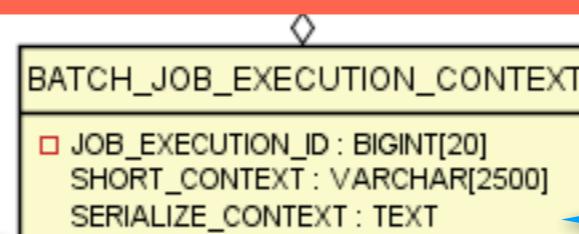
Jobの状態・実行結果
(增加)

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



Stepコンテキストオブジェクトの
シリアルライズデータ



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

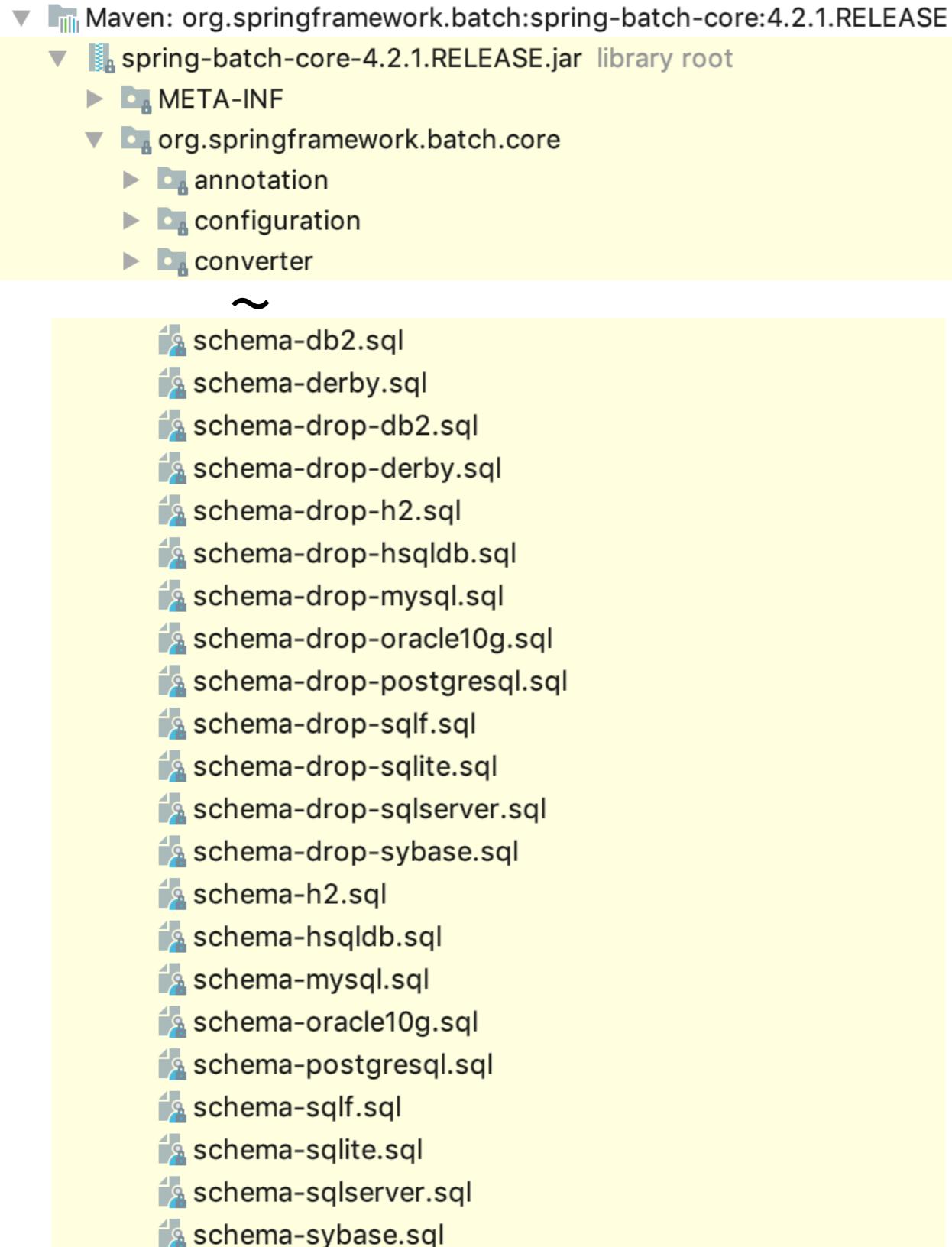
batch_job_executionテーブル (列を抜粋)

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

参考：同じジョブを同じ引数で再実行可能にする

JobParameterConverterをJsrJobParameterConverterにすることで、本事象を回避することが可能

```
@Bean  
public JsrJobParametersConverter jobParametersConverter() {  
    return new JsrJobParametersConverter(dataSource);  
}
```

設定後の実行ログ

```
: Running default command line with: []  
: Job: [SimpleJob: [name=demo01]] launched with the following parameters:  
[{{jsr_batch_run_id=4}]  
: Executing step: [demo01Takslet]  
: test tasklet.  
...  
実行された
```

自動的にジョブ引数が追加され
ジョブ引数がユニークになる

Spring Bootで
Spring Batchを使うべき？

Spring BootでSpring Batchを使う

Spring Batch単体の起動引数

```
$ 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> ...
```

Spring Bootでの追記コードと起動引数

```
@SpringBootApplication  
@EnableBatchProcessing  
public class BatchDemo01Application {  
    public static void main(String[] args) {  
        SpringApplication.run(BatchDemo01Application.class, args);  
    }  
}
```

...

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

Spring BootでSpring Batchを使う

Spring Batch単体の起動引数

```
$ 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> ...
```

複数Jobを持たせる場合は注意
(スキャンで全JobのBeanがロードされる)

Spring Bootでの追記コードと起動引数

```
@SpringBootApplication  
@EnableBatchProcessing  
public class BatchDemo01Application {  
    public static void main(String[] args) {  
        SpringApplication.run(BatchDemo01Application.class, args);  
    }  
}
```

...

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

Spring Batch単体との違い

- Spring Boot機能が使える
 - 実行可能Jar
 - データソースのBean定義が不要
 - 各Starterが使える、等
- デフォルトで全てのJobを実行（ジョブ指定も可能）
- **@BatchDataSource**で複数データソース切替が可能
- コマンドライン引数の影響（--付きパラメータ）

```
$ java -jar myapp.jar --server.port=7070 someParameter=someValue
```

This provides two arguments to the batch job: `someParameter=someValue` and `-server.port=7070`. Note that the second argument is missing one hyphen as Spring Batch will remove the first `-` character of a property if it is present.

<https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#passing-command-line-arguments>

@BatchDataSource (Spring Boot 2.2+)

- JobRepositoryで使うデータソースを、アプリケーションで使うデータソースと別で定義できる (Spring Bootでの制約を緩和)
 - JobRepositoryで使用するデータソースを @BatchDataSource でBean定義
 - アプリケーションで使うデータソースは、@Primaryを付けて別にBean定義
 - アプリケーションで使う側は TransactionManager の作成が必要

```
@Bean  
@BatchDataSource  
public DataSource dsForJobRepos() {  
    return dataSourceForJobRepos;  
}
```

// TransactionManagerは自動生成される

```
@Bean  
@Primary  
public DataSource dsForApp() {  
    return dataSourceForApplication;  
}  
  
@Bean  
public PlatformTransactionManager tmForApp(  
    @Qualifier("dsForApp") DataSource ds) {  
    return new DataSourceTransactionManager(ds);  
}
```

Spring BootでSpring Batchを使うべき？

(個人的な意見)

- Spring Batch単体で見たら必要性は低い
 - Webのようなメリット（組込、Actuator）は少ない
 - 実行可能Jarの簡易作成が“生きれば”アリ
 - 1jar多Jobは向かない（スキャンで全Beanロード）
- Spring Batch以外との兼ね合いで採否を決めるべき
 - 同システムのWebアプリ側に合わせる
 - データアクセスライブラリ連携の有無（Starter）

まとめ

- Tasklet v.s. Chunk
→リカバリ方法や設計難易度を踏まえて決定
- JobRepositoryに関する疑問や注意事項
→DDLや各TBLのキャパシティ考慮
- Spring BootでSpring Batchを使うべき？
→起動引数の違いや1jar多Jobに注意

その他紹介できなかったこと

- Micrometer Metrics (Spring Batch 4.2+)
- Stepの多重実行
- 終了コード周り
- 例外ハンドリング