◎Spring batch

- ジョブスケジューラ

- シエルスクリプト

- Javaをはじめとするプログラム言語

〇タスクレット方式、チャンク方式

〇Spring batchの基本構造

JobLauncher・Job・Step・JobRepository、ItemReader・ItemProcessor・ItemWriter

〇Jobの定義。Truncateステップ

@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();

}

@Bean

public MethodInvokingTaskletAdapter truncateTasklet() {

MethodInvokingTaskletAdapter adapter = new MethodInvokingTaskletAdapter(); ❸

adapter.setTargetObject(truncateService());

adapter.setTargetMethod("execute");

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;

}

}

●ImportFileStep

@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();

fileItemReader.setResource(loader.getResource(filePath));

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

DelimitedLineTokenizer lineTokenizer = new DelimitedLineTokenizer();

lineTokenizer.setNames(new String[]{"roomId", "roomName", "capacity"});

lineMapper.setLineTokenizer(lineTokenizer);

BeanWrapperFieldSetMapper<Room> fieldSetMapper = new BeanWrapperFieldSetMapper<>();

fieldSetMapper.setTargetType(Room.class);

lineMapper.setFieldSetMapper(fieldSetMapper);

fileItemReader.setLineMapper(lineMapper);

fileItemReader.setLinesToSkip(1);

return fileItemReader;

}

@Bean

@StepScope ❹

public ItemWriter<Room> dbItemWriter() {

JpaItemWriter<Room> jpaItemWriter = new JpaItemWriter<>(); ⓫

jpaItemWriter.setEntityManagerFactory(entityManagerFactory);

return jpaItemWriter;

}

}

●テスト

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(classes = {ImportFileConfig.class})

@ActiveProfiles("dev")

public class JobTest {

@Autowired

Job job; ❷

@Autowired

JobLauncher launcher; ❷

@Autowired

JobRepository jobRepository; ❷

@Bean

public JobLauncherTestUtils jobLauncherTestUtils() {

JobLauncherTestUtils utils = new JobLauncherTestUtils();

utils.setJob(job);

utils.setJobLauncher(launcher);

utils.setJobRepository(jobRepository);

return utils;

}

@Test

public void testJob() throws Exception {

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

map.put("filePath", new JobParameter("rooms.csv"));

map.put("executedTime", new JobParameter("201511161000"));

JobParameters params = new JobParameters(map);

BatchStatus status = jobLauncherTestUtils().launchJob(params).getStatus();

assertThat([status.name](http://status.name/)(), is("COMPLETED"));

}

}

〇JOBの起動

●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> ...

〇チャンク方式のXML定義例

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>

〇処理の並行実行

■ 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。