Spring Batch

Mục lục

1.Các khái niệm cơ bản.

1.1. JOB

Job là một thực thể gói gọn toàn bộ quá trình batch

1.2. JobInstance

JobInstance đề cập đến khái niệm về một quy trình công việc hợp lý. Hãy xem xét một công việc hàng loạt sẽ được chạy một lần vào cuối ngày, chẳng hạn như Công việc Cuối ngày từ sơ đồ trước. Có một công việc EndOfDay nhưng mỗi lần chạy Công việc riêng lẻ phải được theo dõi riêng. Trong trường hợp công việc này, có một JobInstance hợp lý mỗi ngày. Ví dụ: có đợt chạy vào ngày 1 tháng 1, đợt chạy vào ngày 2 tháng 1, v.v. Nếu lần chạy ngày 1 tháng 1 không thành công ở lần đầu tiên và được chạy lại vào ngày hôm sau thì đó vẫn là lần chạy ngày 1 tháng 1. (Thông thường, điều này cũng tương ứng với dữ liệu mà nó đang xử lý, nghĩa là lần chạy ngày 1 tháng 1 sẽ xử lý dữ liệu cho ngày 1 tháng 1). Do đó, mỗi JobInstance có thể có nhiều lần thực thi (JobExecution sẽ được thảo luận chi tiết hơn ở phần sau của chương này) và chỉ một JobInstance (tương ứng với một Job cụ thể và xác định JobParameters) có thể chạy tại một thời điểm nhất định.

Định nghĩa của JobInstance hoàn toàn không liên quan gì đến dữ liệu được tải. Việc triển khai ItemReader hoàn toàn phụ thuộc vào việc xác định cách tải dữ liệu. Ví dụ: trong kịch bản EndOfDay, có thể có một cột trên dữ liệu cho biết ngày có hiệu lực hoặc ngày theo lịch trình chứa dữ liệu. Vì vậy, lần chạy ngày 1 tháng 1 sẽ chỉ tải dữ liệu từ lần chạy đầu tiên và lần chạy ngày 2 tháng 1 sẽ chỉ sử dụng dữ liệu từ lần chạy thứ 2. Bởi vì quyết định này có thể là một quyết định kinh doanh nên nó được để cho ItemReader quyết định. Tuy nhiên, việc sử dụng cùng một JobInstance sẽ xác định xem "trạng thái" (tức là ExecutionContext, sẽ được thảo luận sau trong chương này) từ các lần thực thi trước đó có được sử dụng hay không. Sử dụng JobInstance mới có nghĩa là “bắt đầu lại từ đầu” và sử dụng phiên bản hiện có thường có nghĩa là “bắt đầu từ nơi bạn đã dừng lại”.

1.3 JobParameter

JobInstance = Job + identifying JobParameters Điều này cho phép nhà phát triển kiểm soát hiệu quả cách xác định JobInstance vì họ kiểm soát những tham số nào được truyền vào.

1.4 JobExcution

JobExecution đề cập đến khái niệm kỹ thuật về một nỗ lực duy nhất để thực hiện một Công việc. Việc thực thi có thể kết thúc thành công hoặc thất bại, nhưng JobInstance tương ứng với một lần thực thi nhất định không được coi là hoàn thành trừ khi việc thực thi hoàn thành thành công. Sử dụng Công việc EndOfDay được mô tả trước đây làm ví dụ, hãy xem xét một JobInstance cho ngày 01-01-2017 không thành công trong lần chạy đầu tiên. Nếu nó được chạy lại với cùng tham số xác định công việc như lần chạy đầu tiên (01-01-2017), một JobExecution mới sẽ được tạo. Tuy nhiên, vẫn chỉ có một JobInstance.

Job xác định công việc là gì và nó được thực thi như thế nào, còn JobInstance là một đối tượng có tổ chức thuần túy để nhóm các lần thực thi lại với nhau, chủ yếu là để cho phép ngữ nghĩa khởi động lại chính xác. Tuy nhiên, JobExecution là cơ

chế lưu trữ chính cho những gì thực sự xảy ra trong quá trình chạy và chứa nhiều thuộc tính khác phải được kiểm soát và duy trì,

- Batch Status: cho biết trạng thái của Batch
- startTime : Thời gian bắt đầu, trống nếu job chưa start
- endTime: Thời gian kết thúc, trống nếu job chưa finish.
- exitStatus: Thoát và trả về kết quả cho người gọi, trống nếu job chưa finish.
- createTime: Thời gian tạo job.
- lastUpdate: Thời gian cuối update, trống nếu job chưa start.
- excutionContren: Chứa nội dung dữ liệu user
- failureException: Danh sách ngoại lệ trong quá trình thực hiện Job.

1.5 STEP

Step là một đối tượng miền bao gồm một giai đoạn tuần tự, độc lập của một công việc hàng loạt. Do đó, mọi Công việc đều bao gồm toàn bộ một hoặc nhiều bước. Bước chứa tất cả thông tin cần thiết để xác định và kiểm soát quá trình xử lý hàng loạt thực tế. Đây nhất thiết phải là một mô tả mơ hồ vì nội dung của bất kỳ Bước cụ thể nào đều do nhà phát triển viết Công việc quyết định. Một Bước có thể đơn giản hoặc phức tạp tùy theo mong muốn của nhà phát triển. Một Bước đơn giản có thể tải dữ liệu từ một tệp vào cơ sở dữ liệu, yêu cầu ít hoặc không cần mã (tùy thuộc vào cách triển khai được sử dụng). Một Bước phức tạp hơn có thể có các quy tắc kinh doanh phức tạp được áp dụng như một phần của quá trình xử lý. Giống như một Công việc, một Bước có một StepExecution riêng lẻ tương quan với một JobExecution duy nhất.

StepExecution thể hiện một nỗ lực duy nhất để thực hiện một Bước. Một StepExecution mới được tạo mỗi khi một Bước được chạy, tương tự như JobExecution. Tuy nhiên, nếu một bước không thực thi được vì bước trước đó không thành công thì bước đó sẽ không được thực thi. StepExecution chỉ được tạo khi Bước của nó thực sự được bắt đầu.
Việc thực thi các bước được thể hiện bằng các đối tượng của lớp StepExecution. Mỗi lần thực thi chứa một tham chiếu đến bước tương ứng của nó, JobExecution và dữ liệu liên quan đến giao dịch, chẳng hạn như số lần cam kết và khôi phục cũng như thời gian bắt đầu và kết thúc. Ngoài ra, mỗi bước thực thi đều chứa ExecutionContext, chứa mọi dữ liệu mà nhà phát triển cần duy trì trong suốt các lần chạy hàng loạt, chẳng hạn như số liệu thống kê hoặc thông tin trạng thái cần thiết để khởi động lại.

- readCount: Số item đọc thành công.
- writeCount: Số item viết thành công.
- commitCount: Số giao dịch được commit cho execution
- rollbackCount: Số giao dịch kinh doanh được điều khiển bởi Step đã được roll back.
- readSkipCount: Số lần đọc thất bại, kết quả trong skipped item.
- processSkipCount: Số lần thực thi thất bại, kết quả trong skipped item.
- filterCount: Số item được lọc bởi ItemProcessor.
- writeSkipCount: Số lần ghi thất bại, kết quả trong skipped item.

1.6 JobRepository

JobRepository lưu trữ metadata của các công việc (jobs) được chạy bởi Spring Batch.

Nó cung cấp CRUD cho JobLaunched, Job và Step

2. Job Configution.

Configuring a Job

There are multiple implementations of the `Job` interface. However, these implementations are abstracted behind either the provided builders (for Java configuration) or the XML namespace (for XML-based configuration). The following example shows both Java and XML configuration:

Java XML

JAVA

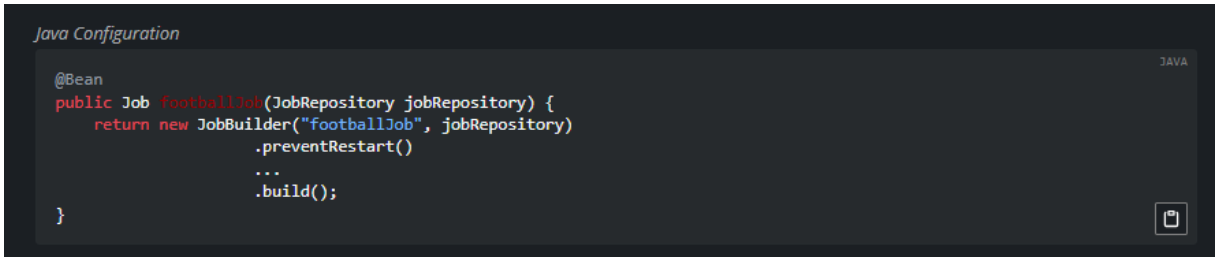
```
@Bean
public Job footballJob(JobRepository jobRepository) {
    return new JobBuilder("footballJob", jobRepository)
                     .start(playerLoad())
                     .next(gameLoad())
                     .next(playerSummarization())
                     .build();
}
```

A `Job` (and, typically, any `Step` within it) requires a `JobRepository`. The configuration of the `JobRepository` is handled through the `Java Configuration`.

The preceding example illustrates a `Job` that consists of three `Step` instances. The job related builders can also contain other elements that help with parallelization (`Split`), declarative flow control (`Decision`), and externalization of flow definitions (`Flow`).

2.1 Restartability(Khả năng tái khởi động)

Một vấn đề chính khi thực hiện một công việc hàng loạt liên quan đến hành vi của Công việc khi nó được khởi động lại. Việc khởi chạy một Job được coi là "khởi động lại" nếu JobExecution đã tồn tại cho JobInstance cụ thể. Lý tưởng nhất là tất cả các công việc đều có thể bắt đầu từ nơi chúng đã dừng lại, nhưng có những trường hợp điều này là không thể. Trong trường hợp này, việc đảm bảo rằng JobInstance mới được tạo hoàn toàn phụ thuộc vào nhà phát triển. Tuy nhiên, Spring Batch có cung cấp một số trợ giúp. Nếu Công việc không bao giờ được khởi động lại nhưng phải luôn được chạy như một phần của Phiên bản công việc mới, thì bạn có thể đặt thuộc tính có thể khởi động lại thành false.



Java Configuration

JAVA

```
@Bean
public Job footballJob(JobRepository jobRepository) {
    return new JobBuilder("footballJob", jobRepository)
                     .preventRestart()
                     ...
                     .build();
}
```

Nói cách khác, cài đặt có thể khởi động lại thành sai có nghĩa là “Công việc này không hỗ trợ bắt đầu lại”. Việc khởi động lại một Công việc không thể khởi động lại sẽ khiến cho ngoại lệ JobRestartException bị ném ra. Mã Junit sau đây gây ra ngoại lệ:

```java
Job job = new SimpleJob();
job.setRestartable(false);

JobParameters jobParameters = new JobParameters();

JobExecution firstExecution = jobRepository.createJobExecution(job, jobParameters);
jobRepository.saveOrUpdate(firstExecution);

try {
    jobRepository.createJobExecution(job, jobParameters);
    fail();
}
catch (JobRestartException e) {
    // expected
}
```

2.2 Intercepting job execution(Đánh chặn thực hiện job)

During the course of the execution of a `Job`, it may be useful to be notified of various events in its lifecycle so that custom code can be run. `SimpleJob` allows for this by calling a `JobListener` at the appropriate time:

```java
public interface JobExecutionListener {

    void beforeJob(JobExecution jobExecution);

    void afterJob(JobExecution jobExecution);
}
```

You can add `JobListeners` to a `SimpleJob` by setting listeners on the job.

You can add `JobListeners` to a `SimpleJob` by setting listeners on the job.

Java | XML

The following example shows how to add a listener method to a Java job definition:

Java Configuration

```java
@Bean
public Job footballJob(JobRepository jobRepository) {
    return new JobBuilder("footballJob", jobRepository)
                     .listener(sampleListener())
                     ...
                     .build();
}
```

Note that the `afterJob` method is called regardless of the success or failure of the `Job`. If you need to determine success or failure, you can get that information from the `JobExecution`:

```java
public void afterJob(JobExecution jobExecution){
    if (jobExecution.getStatus() == BatchStatus.COMPLETED ) {
        //job success
    }
    else if (jobExecution.getStatus() == BatchStatus.FAILED) {
        //job failure
    }
}
```

The annotations corresponding to this interface are:

- `@BeforeJob`
- `@AfterJob`

2.3 Kế thừa từ Job bố mẹ

Giống trong Java

3. Configuring a Step.

3.1 Cấu hình Step

Java Configuration

```java
/**
 * Note the JobRepository is typically autowired in and not needed to be explicitly
 * configured
 */
@Bean
public Job sampleJob(JobRepository jobRepository, Step sampleStep) {
    return new JobBuilder("sampleJob", jobRepository)
                .start(sampleStep)
                .build();
}

/**
 * Note the TransactionManager is typically autowired in and not needed to be explicitly
 * configured
 */
@Bean
public Step sampleStep(JobRepository jobRepository, 2
        PlatformTransactionManager transactionManager) { 1
    return new StepBuilder("sampleStep", jobRepository)
                .<String, String>chunk(10, transactionManager) 3
                .reader(itemReader())
                .writer(itemWriter())
                .build();
}
```

transactionManager: PlatformTransactionManager của Spring bắt đầu và thực hiện các giao dịch trong quá trình xử lý.

Repository: Tên dành riêng cho Java của kho lưu trữ JobRepository định nghĩa StepExecution và ExecutionContext trong quá trình xử lý (ngay trước khi cam kết).

chunk : cho biết đây là bước dựa trên mục và số lượng mục cần xử lý trước khi giao dịch được thực hiện.

3.2 Kế thừa từ Step bố mẹ

Kế thừa giống trong java.

3.3 The Commit Interval

`.<String, String>chunk(10, transactionManager)`

Số item được xử lí trong 1 transaction. ở đây là 10.

3.4 Cấu hình một Step tái khởi động

Việc khởi động lại có nhiều tác động đến các bước và do đó, có thể yêu cầu một số cấu hình cụ thể.

3.4.1 Setting a start limit:

Có nhiều tình huống mà bạn có thể muốn kiểm soát số lần một Bước có thể được bắt đầu. Ví dụ: bạn có thể cần định cấu hình một Bước cụ thể để nó chỉ chạy một lần vì nó làm mất hiệu lực một số tài nguyên phải được sửa thủ công trước khi có thể chạy lại. Điều này có thể được cấu hình ở cấp độ bước vì các bước khác nhau có thể có các yêu cầu khác nhau. Một Bước chỉ có thể được thực hiện một lần có thể tồn tại như một phần của cùng một Job như một Step có thể chạy vô hạn.

Java Configuration

JAVA

```java
@Bean
public Step step1(JobRepository jobRepository, PlatformTransactionManager transactionManager) {
    return new StepBuilder("step1", jobRepository)
                .<String, String>chunk(10, transactionManager)
                .reader(itemReader())
                .writer(itemWriter())
                .startLimit(1)
                .build();
}
```

Giới hạn step ở đây là chạy 1 lần

3.4.2 Tái khởi động khi một Step hoàn thành

Trong các trường hợp công việc có thể khởi động lại, có thể có một hoặc nhiều bước phải luôn được chạy, bất kể chúng có thành công trong lần đầu tiên hay không. Một ví dụ có thể là bước xác thực hoặc Bước làm sạch tài nguyên trước khi xử lý. Trong quá trình xử lý thông tin thường xuyên của một công việc được khởi động lại, bất kỳ bước nào có trạng thái HOÀN THÀNH (có nghĩa là nó đã được hoàn thành thành công) đều bị bỏ qua. Đặt allow-start-if-complete thành true sẽ ghi đè điều này để bước này luôn chạy

Java Configuration

JAVA

```java
@Bean
public Step step1(JobRepository jobRepository, PlatformTransactionManager transactionManager) {
    return new StepBuilder("step1", jobRepository)
                .<String, String>chunk(10, transactionManager)
                .reader(itemReader())
                .writer(itemWriter())
                .allowStartIfComplete(true)
                .build();
}
```

3.5 Configuring Skip Logic

Có nhiều trường hợp trong đó lỗi gặp phải trong khi xử lý sẽ không dẫn đến lỗi Bước mà thay vào đó nên bỏ qua. Đây thường là quyết định phải được đưa ra bởi người hiểu rõ dữ liệu và ý nghĩa của nó. Ví dụ: dữ liệu tài chính không thể bỏ qua vì nó dẫn đến việc chuyển tiền và dữ liệu này cần phải hoàn toàn chính xác. Mặt khác, việc tải danh sách các nhà cung cấp có thể cho phép bỏ qua. Nếu nhà cung cấp không được tải vì định dạng không chính xác hoặc thiếu thông tin cần thiết thì có thể không có vấn đề gì. Thông thường, những bản ghi xấu này cũng được ghi lại, điều này sẽ được đề cập sau khi thảo luận về người nghe.

Sử dụng lệnh limit

3.6 Configuring Retry Logic

Trong hầu hết các trường hợp, bạn muốn có một ngoại lệ gây ra lỗi bỏ qua hoặc Bước. Tuy nhiên, không phải tất cả các trường hợp ngoại lệ đều mang tính quyết định. Nếu gặp phải FlatFileParseException trong khi đọc, nó sẽ luôn được ném ra cho bản ghi đó. Đặt lại ItemReader không giúp ích gì. Tuy nhiên, đối với các trường hợp ngoại lệ khác (chẳng hạn như DeadlockLoserDataAccessException, cho

biết quy trình hiện tại đã cố cập nhật bản ghi mà quy trình khác đang khóa), việc chờ và thử lại có thể dẫn đến thành công.

Sử dụng lệnh retry

3.7 Controlling Rollback

. Theo mặc định, bất kể thử lại hay bỏ qua, bất kỳ trường hợp ngoại lệ nào được đưa ra từ ItemWriter đều khiến giao dịch do Bước kiểm soát bị khôi phục. Nếu bỏ qua được định cấu hình như mô tả trước đó, các ngoại lệ được gửi từ ItemReader sẽ không gây ra quá trình khôi phục. Tuy nhiên, có nhiều trường hợp trong đó các ngoại lệ được đưa ra từ ItemWriter sẽ không gây ra hiện tượng khôi phục vì chưa có hành động nào được thực hiện để vô hiệu hóa giao dịch. Vì lý do này, bạn có thể định cấu hình Bước với danh sách các trường hợp ngoại lệ sẽ không gây ra sự quay lui.

Sử dụng lệnh noRollback

4. ItemReaders và ItemWriters

4.1 ItemReader

ItemReader là phương tiện cung cấp dữ liệu từ nhiều loại đầu vào khác nhau

4.2 ItemWriter

ItemReader là phương tiện cung cấp dữ liệu từ nhiều loại đầu ra khác nhau

4.3 ItemStream:

Cả ItemReaders và ItemWriters đều phục vụ tốt các mục đích riêng của họ, nhưng cả hai đều có mối lo ngại chung là cần có giao diện khác. Nói chung, là một phần trong phạm vi của công việc hàng loạt, trình đọc và trình ghi cần được mở, đóng và yêu cầu cơ chế duy trì trạng thái. Giao diện ItemStream phục vụ mục đích đó

4.4 Flat Files

4.4.1 FieldSet

name as patterned after `ResultSet`, as shown in the following example:

```java
String[] tokens = new String[]{"foo", "1", "true"};
FieldSet fs = new DefaultFieldSet(tokens);
String name = fs.readString(0);
int value = fs.readInt(1);
boolean booleanValue = fs.readBoolean(2);
```

4.4.2 FlatFileItemReader

+Xác định FieldSetMapper để ánh xạ

```
ublic class InformationReader implements ItemReader<String> {

    protected static class InformationFieldSetMapper implements FieldSetMapper<Information>{
        public Information mapFieldSet(FieldSet fieldSet) {
            Information information = new Information();
            information.setName(fieldSet.readString(0));
            information.setAge(fieldSet.readInt(1));
            information.setAvgMark(fieldSet.readDouble(2));
            return information;
        }
    }
```

+Đọc file csv

```
    @Override
    public String read()
            throws Exception, UnexpectedInputException, ParseException, NonTransientResourceException {

        FlatFileItemReader<Information> itemReader = new FlatFileItemReader<Information>();
        itemReader.setLinesToSkip(index);
        itemReader.setResource(new FileSystemResource("G:\\Job\\SpringBatch\\src\\main\\resources\\read.csv"));
        DefaultLineMapper<Information> lineMapper = new DefaultLineMapper<Information>();
        lineMapper.setLineTokenizer(new DelimitedLineTokenizer());
        lineMapper.setFieldSetMapper(new InformationFieldSetMapper());
        itemReader.setLineMapper(lineMapper);
        itemReader.open(new ExecutionContext());
        Information information = itemReader.read();
        System.out.println("Name" +  information.getName());

        if(index >= 4) {
            return null;
        }
        index++;
        return information.getName();
    }
```

4.4.3 FlatFileItemWriter

+Simplified File Writing

LineAggregator: Ghi một đối tượng thành chuỗi

```
        return new FlatFileItemWriterBuilder<Information>()
                .name("itemwriter")
                .resource(new FileSystemResource("G:\\Job\\SpringBatch\\src\\main\\resources\\test.txt"))
                .lineAggregator(new PassThroughLineAggregator<Information>())
                .build();
    }
```

PassThroughLineAggregator

là triển khai của phổ biến nhất của LineAggregator để biến một đối tượng thành một chuỗi

+Delimited File Writing

```
    @Bean
    public FlatFileItemWriter<Information> itemWriter3(Resource resource){
        return new FlatFileItemWriterBuilder<Information>()
                .name("InformationWriter3")
                .resource((WritableResource) resource)
                .delimited()
                .delimiter("|")
                .names(new String[] {"name" , "age" , "avgMark" , "classfication"})
                .build();
    }
```

Định dạng khoảng rộng khi writer

```
FormatterLineAggregator<CustomerCredit> lineAggregator = new FormatterLineAggregator<>();
lineAggregator.setFormat("%-9s%-2.0f");
lineAggregator.setFieldExtractor(fieldExtractor);
```

Java Configuration

JAVA

```
@Bean
public FlatFileItemWriter<CustomerCredit> itemWriter(Resource outputResource) throws Exception {
    return new FlatFileItemWriterBuilder<CustomerCredit>()
                .name("customerCreditWriter")
                .resource(outputResource)
                .formatted()
                .format("%-9s%-2.0f")
                .names(new String[] {"name", "credit"})
                .build();
}
```

4.5. Json item reader and writer

4.5.1 reader

```
@Bean
public JsonItemReader<Trade> itemReaderJson(){
    return new JsonItemReaderBuilder<Trade>()
            .jsonObjectReader(new JacksonJsonObjectReader<Trade>(Trade.class))
            .resource(new FileSystemResource("G:\\Job\\SpringBatch\\src\\main\\resources\\read.json"))
            .name("readerJson")
            .build();

}
```

4.5.2 Writer

```
@Bean JsonFileItemWriter<Trade> itemWriterJson(){
    return new JsonFileItemWriterBuilder<Trade>()
            .jsonObjectMarshaller(new JacksonJsonObjectMarshaller<Trade>())
            .resource(new FileSystemResource("G:\\Job\\SpringBatch\\src\\main\\resources\\write.json"))
            .name("writerJson")
            .build();
}
```

4.6. Multi-File Input

```
private Resource[] inputResources = {new FileSystemResource("G:\\Job\\SpringBatch\\src\\main\\resources\\Read.json"),
        new FileSystemResource("G:\\Job\\SpringBatch\\src\\main\\resources\\Read2.json")};
@Bean
public MultiResourceItemReader<Trade> multiResourceItemReader(){
    return new MultiResourceItemReaderBuilder<Trade>()
            .name("multiFile")
            .resources(inputResources)
            .delegate(itemReaderJson())
            .build();
}
```

```
@Bean
public Step step2(JobRepository jobRepository , PlatformTransactionManager transactionManager) throws ClassNotFoundException  {
    return new StepBuilder("step2" , jobRepository)
            .<Trade, Trade> chunk(3,transactionManager)
            .reader(multiResourceItemReader())
            .processor(new TradeProcessor())
            .writer(itemWriterJson())
            .build();
}
```

4.7 Database

4.7.1 Database input

```
    }
    @Autowired
    private DataSource dataSource;

    @Bean
    public JdbcCursorItemReader<Information> itemReaderJDBC(){
        return new JdbcCursorItemReaderBuilder<Information>()
                .dataSource(dataSource)
                .name("InformationReader")
                .sql("Select ID, NAME,AGE, AVG_MARK ,CLASSFICATION from Information")
                .rowMapper(new InformationRowMapper())
                .build();
    }
```

4.7.2 Database output

```
public class InformationWriter implements ItemWriter<Information> {

    private final InformationRepository informationRepository;

    @Autowired
    public InformationWriter(InformationRepository informationRepository) {
        this.informationRepository = informationRepository;
    }

    @Override
    public void write(Chunk<? extends Information> chunk) throws Exception {

        for(Information information : chunk) {
            System.out.println("Save in Database");
            informationRepository.save(information);
        }
        System.out.println("Complete Writer");

    }

}
```