Spring Batch

--Created by David Song

Contents

Foreword

Spring Batch is a framework for batch processing, with rich features. This document will illustrate a few examples to capture the most useful features.

Introduction

A batch processing is for those frequently used programs that can be executed with minimal human interaction, for example, scheduled batch jobs run by CRON command in Unix. A program that reads a large file and generates a report, for example, is considered to be a batch job.

Spring Batch is a framework designed to enable the development of batch applications. Each batch process can be viewed as a job, which has a few steps; and each step can have multiple tasks. Spring Batch framework provide a JobRepositiory which stored jobs metadata, a JobLauncher to run robs, in order to stored jobs metadata.

Spring Batch provides two implementations of the `JobRepositiory` interface: one stores metadata in memory, which is useful for testing or when you don't want monitoring or restart capabilities; the other stored metadata in a relational database.

Spring Batch provides a job repository implementation to store job metadata in a database. This allows you to monitor the execution of your batch processes and their results status.

Simple batch job without DB

This section will show a simple example how to create and launch a spring batch job without using database. The example is twisted from [2]. Code can be found at [4]

Please note that, MapJobRepositoryFactoryBean is deprecated in Spring Batch 4 and removed from version 5, need to replace it with an equivalent when using Spring Batch 5.

Code structure:
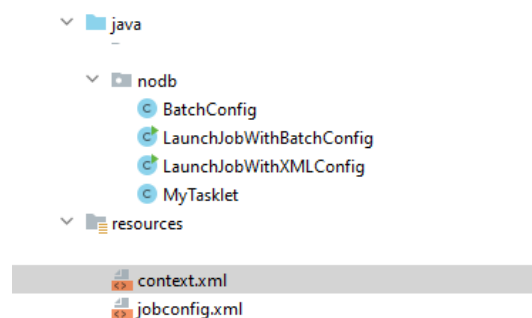
With xml configuration

Context.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">

<bean id="jobRepository"
      class="org.springframework.batch.core.repository.support.MapJobRepositoryFactoryBean">
    <property name="transactionManager" ref="transactionManager"/>
</bean>

<bean id="transactionManager"
      class="org.springframework.batch.support.transaction.ResourcelessTransactionManager"/>

<bean id="jobLauncher"
      class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
    <property name="jobRepository" ref="jobRepository"/>
</bean>

</beans>
```

Jobconfig.xml

```xml
<beans xmlns:batch="http://www.springframework.org/schema/batch"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/batch
      http://www.springframework.org/schema/batch/spring-batch-2.2.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.2.xsd ">

    <import resource="context.xml"/>

    <!-- Defining a bean -->
    <bean id="tasklet" class="nodb.MyTasklet"/>

    <!-- Defining a job-->
    <batch:job id="helloWorldJob">
        <!-- Defining a Step -->
        <batch:step id="step1">
            <tasklet ref="tasklet"/>
        </batch:step>
    </batch:job>
</beans>
```

Launch Job

```java
package nodb;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.JobParameters;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class LaunchJobWithXMLConfig {
    public static void main(String[] args) throws Exception {

        System.out.println("main started ...");

        //
        // load the configuration
        //
```

```
        String[] springConfig  =  {"jobconfig.xml"};
        ApplicationContext context = new ClassPathXmlApplicationContext(springConfig);

        JobLauncher jobLauncher = (JobLauncher) context.getBean("jobLauncher");
        Job job = (Job) context.getBean("helloWorldJob");

        //
        // Executing the JOB with no parameters
        //
        JobExecution execution = jobLauncher.run(job, new JobParameters());
        System.out.println("Exit Status : " + execution.getStatus());
    }
}
```

Java bean configuration

```
package nodb;

import …

@Configuration
public class BatchConfig {

    @Bean
    ResourcelessTransactionManager transactionManager(){
        return new ResourcelessTransactionManager();
    }

    @Bean
    JobRepository jobRepository(){
        try {
            return new MapJobRepositoryFactoryBean(transactionManager()).getObject();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    @Bean(name = "jobLauncher")
    SimpleJobLauncher jobLauncher(){
        SimpleJobLauncher jobLauncher =  new SimpleJobLauncher();
        jobLauncher.setJobRepository(jobRepository());
        return jobLauncher;
    }

    @Bean
    Job helloWorldJob(){

        JobBuilderFactory factory = new JobBuilderFactory(jobRepository());

        JobBuilder builder = factory.get("helloWorldJob");
        return builder.flow(step1()).end().build();

    }
    @Bean
    public Step step1() {
        StepBuilderFactory stepFactory = new StepBuilderFactory(jobRepository(),
transactionManager());
        return stepFactory.get("step1").tasklet(tasklet()).build();
    }

    @Bean
    MyTasklet tasklet(){
        return new MyTasklet();
    }

}
```

Launch the job

```
package nodb;
import ……

public class LaunchJobWithBatchConfig {
    public static void main(String[] args) throws Exception {

        System.out.println("main started ...");

        //
        // load the configuration
        //
        ApplicationContext context = new AnnotationConfigApplicationContext(BatchConfig.class);
        String[] beans = context.getBeanDefinitionNames();
        for (String bean : beans) {
            System.out.println(bean);
        }

        JobLauncher jobLauncher = (JobLauncher) context.getBean("jobLauncher");
        Job job = (Job) context.getBean("helloWorldJob");

        //
        // Executing the JOB with no parameters
        //
        JobExecution execution = jobLauncher.run(job, new JobParameters());
        System.out.println("Exit Status : " + execution.getStatus());
    }
}
```

From above examples, we can see that the xml configuration is easier and clearer than the java bean configuration in this case.

Simple batch job with DB

Embedded Database

An embedded database is an in memory database, it exists only during your application running state. There are a few Java embedded database available, written in Java, such as like HSQL, H2 and Derby.

The embedded database concept is very helpful during the development and testing phases. A heavy weight database such as Oracle, MS SQL server, are normally shared between team members, it's easy to cause conflict of resources. Using in memory database erases the headache during development phase.

Below shows the way to start a H2 database, insert data, and inspect the database from a HSQL popup window and H2 server

```
public static void main(String[] args) {

    // start H2
    new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.H2)
            .setName("testdb")                        // optional, default is testdb
            .addScript("db/sql/create-db.sql")        // create tables in
            .addScript("db/sql/insert-data.sql")      // insert some data
            .build();

    // hsqldb popups a Java Swing window
    DatabaseManagerSwing.main(new String[]{"--url", "jdbc:h2:mem:testdb", "--user", "sa", "--
```

```java
password", ""});

    // Start H2 webserver, http://localhost:8082/
    try {
        Server.createWebServer("-web", "-webAllowOthers", "-webPort", "8082").start();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }

}
```

Using MySQL

mysql-datebase.xml:

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
   http://www.springframework.org/schema/jdbc
   http://www.springframework.org/schema/jdbc/spring-jdbc-3.2.xsd">

    <!-- MySQL datasource -->
    <bean id="dataSource"
          class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/batch" />
        <property name="username" value="admin" />
        <property name="password" value="passw0rd" />
    </bean>

    <bean id="transactionManager"
          class="org.springframework.batch.support.transaction.ResourcelessTransactionManager" />

    <!-- create job-meta tables -->
    <!-- only use once during init, otherwise all tables will be wiped out
    <jdbc:initialize-database data-source="dataSource">
        <jdbc:script location="org/springframework/batch/core/schema-drop-mysql.sql" />
        <jdbc:script location="org/springframework/batch/core/schema-mysql.sql" />
    </jdbc:initialize-database>
    -->

</beans>
```

mysql-context.xml:
```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
   http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">


    <bean id="jobRepository"
          class="org.springframework.batch.core.repository.support.JobRepositoryFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="transactionManager" ref="transactionManager" />
        <property name="databaseType" value="mysql" />
    </bean>

    <bean id="jobLauncher"
          class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
        <property name="jobRepository" ref="jobRepository" />
    </bean>

</beans>
```

Java code to lunch the job:

```java
public class RunMySQLJob {

    public static void main(String[] args) {

        String[] springConfig  =
                {
                        "mysql-context.xml",
                        "mysql-database.xml",
                        "load-stock-holdings-job-context.xml"
                };

        ApplicationContext context =
                new ClassPathXmlApplicationContext(springConfig);

        JobLauncher jobLauncher = (JobLauncher) context.getBean("jobLauncher");
        Job importStocksJob = (Job) context.getBean("importStocks");

        JobParameters jobParameters = new JobParametersBuilder()
        .addString("targetDirectory", "input//")
        .addString("targetFile","current_holdings.csv")
        .addLong("time",System.currentTimeMillis()) // enable multiple runs
        .toJobParameters();

        try {

            JobExecution execution = jobLauncher.run(importStocksJob, jobParameters);
            System.out.println("Exit Status : " + execution.getStatus());

        } catch (Exception e) {
            e.printStackTrace();
        }

        System.out.println("Done");

    }
```
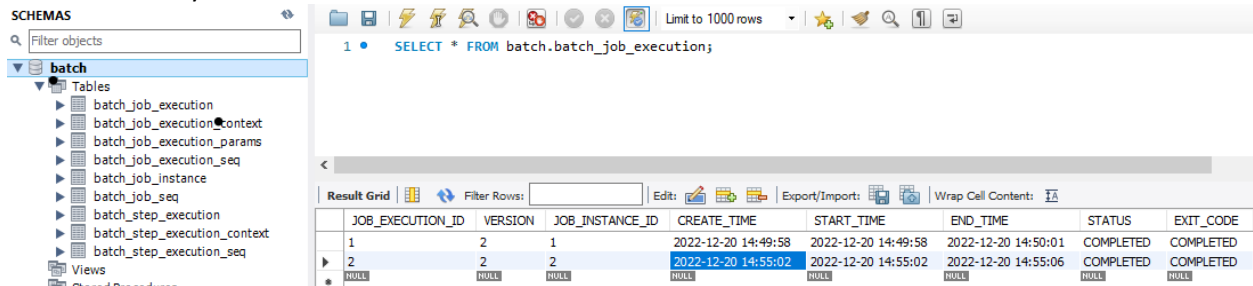
meta data in MySQL database:

Spring Boot Batch

It is common to use batch process within a web server. Spring boot is a good candidate to host batch processing jobs. Code can be found at [5]

Below examples demonstrate:

1. Use H2 as the Batch metadata database
2. Read data from a csv file and write to MySQL database
3. A reader reads data from a csv file, file name and location are parsed from job argument. The reader's beforeStep method reads file header to determine the parsing columns. Other parameters such as skip number of lines, required header etc. can be passed in job parameters. These parsed information in reader's beforeStep method can be put into the stepExecution and can be taken from processor and writer.
4. A processor simply forwards the input from readers to writer.
5. A writer uses a JDBC template to write data to a MySQL database table.
6. Run batch job in multi-thread
7. Partition input

Generic Batch process

In this example, we will use H2 to store Batch Job metadata, and MySQL to store imported data. The goal is to demonstrate how to use different data sources. The application.properties file below shows how to configure data sources.

The application.properties file

```
#
# H2 DB configuration as defalut datasource for batch
#
#spring.datasource.url=jdbc:h2:file:./DB
spring.jpa.properties.hibernate.hbm2ddl.auto=update
spring.batch.job.enabled=false
spring.batch.initialize-schema=always
spring.datasource.url=jdbc:h2:mem:testdb


#
# mysql configuration as default for batch
#
#spring.jpa.hibernate.ddl-auto=update
#spring.datasource.url=jdbc:mysql://localhost:3306/batch
#spring.datasource.username=admin
#spring.datasource.password=passw0rd
#spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
#spring.batch.job.enabled=false

#
# secodary database for data update
#
spring.datasource.mysql.url=jdbc:mysql://localhost:3306/stocks
spring.datasource.mysql.username=admin
spring.datasource.mysql.password=passw0rd
spring.datasource.mysql.driver-class-name=com.mysql.cj.jdbc.Driver
```

The configuration file, shows how to create a job and steps, we use a chunk step here, listeners are just used to illustrate how to setup listeners.

```
@Configuration
public class BatchConfig {

    @Autowired
    public JobBuilderFactory jobBuilderFactory;
```

```java
    @Autowired
    public StepBuilderFactory stepBuilderFactory;

    @Autowired
    JdbcTemplate mysqlJdbcTemplate;

    @Bean
    public Job importStocks() {
                return jobBuilderFactory.get("importStocks")
                            .incrementer(new RunIdIncrementer()).listener(listener())
                            .flow(readWriteStocks()).end().build();
    }

    @Bean
    public Step readWriteStocks() {
                return stepBuilderFactory.get("readWriteStocks")
                            .listener(stepExecutionListener())
                            .<String, String> chunk(1)
                            .listener(chunkItemReadListener(null))
                            .reader(new GenericReader()).processor(new
GenericProcessor())
                            .writer(new GenericWriter(mysqlJdbcTemplate)).build();
    }

    @Bean
    public JobExecutionListener listener() {
                return new JobCompletionListener();
    }

    @Bean
StepExecutionListener stepExecutionListener(){
                return new StepExecutionListener() {
                        @Override
                        public void beforeStep(StepExecution stepExecution) {
                                String fileName = (String)
stepExecution.getJobExecution().getJobParameters()
                                                .getString("fileName");
                                System.out.println("in STEP listener, job param fileName="
+ fileName);

                                ClassLoader classLoader = getClass().getClassLoader();
                                URL resource = classLoader.getResource(fileName);
                                File file = new File(resource.getFile());
                                System.out.println(file.getPath());
                                System.out.println(String.format("File %s
exist:%s",fileName, file.exists()));

                        }

                        @Override
                        public ExitStatus afterStep(StepExecution stepExecution) {
                                System.out.println("in STEP listener, jafterStep");
                                if (stepExecution.getStatus() == BatchStatus.COMPLETED) {
                                        return ExitStatus.COMPLETED;
                                }
                                return ExitStatus.FAILED;
                        }
                };
    }


    @Bean
    @StepScope
    public ItemReadListener chunkItemReadListener(final @Value("#{jobParameters['name']}")
String name) {

                return new ItemListenerSupport() {
                        @Override
                        public void beforeRead() {
                                System.out.println("in listener, job param name=" + name);
                                super.beforeRead();
                        }
```

```
                };
        }

}
```

Reader

```java
public class GenericReader implements ItemReader<String> {

        BufferedReader br;
        int skip_line;
        int count;

        @BeforeStep
        public void beforeStep(StepExecution stepExecution) {
                //
                // get some job parameters
                //
                JobParameters jobParameters =
stepExecution.getJobExecution().getJobParameters();
                String fileName = (String)jobParameters.getString("fileName");
                skip_line = 1;
                count = 0;

                System.out.println("in Reader before step, job param fileName=" + fileName);
                ClassLoader classLoader = getClass().getClassLoader();
                URL resource = classLoader.getResource(fileName);
                File file = new File(resource.getFile());
                System.out.println(file.getPath());
                System.out.println(String.format("File %s exist:%s",fileName, file.exists()));
                try {
                        br = new BufferedReader(new FileReader(file));
                        for(int i=0; i<skip_line;i++){
                                System.out.println("Skip:"+br.readLine());
                        }

                } catch (FileNotFoundException e) {
                        throw new RuntimeException(e);
                } catch (IOException e) {
                        throw new RuntimeException(e);
                }
        }

        //
        // Since we use new operator to create a new reader in job configuration, sync is
unnecessary
        // otherwise, try to set bean cope to prototype, in this way, it will create a new bean
        //
        @Override
        public synchronized String read() throws Exception, UnexpectedInputException,
                        ParseException, NonTransientResourceException {

                long threadId = Thread.currentThread().getId();

                String sCurrentLine;
                while ((sCurrentLine = br.readLine()) != null){
                        count++;

                        System.out.println(String.format("Thread %d: reading: %s", threadId,
sCurrentLine));

                        return sCurrentLine;
                }

                return null;
        }

}
```

Writer

```java
public class GenericWriter implements ItemWriter<String> {

        JdbcTemplate mysqlJdbcTemplate;

        private String table_name;

        private String[] header;
        private int header_length= 0;

        public GenericWriter(JdbcTemplate mysqlJdbcTemplate){
                this.mysqlJdbcTemplate = mysqlJdbcTemplate;
        }

        @BeforeStep
        public void beforeStep(StepExecution stepExecution) {
                //
                // get some job parameters, hardcoded for demo
                //
                JobParameters jobParameters =
stepExecution.getJobExecution().getJobParameters();
                String outfileName = (String) jobParameters.getString("outfileName");
                this.table_name="stock_holdings";
                String[] table_header =
{"ID","SYMBOL","NAME","LASTSALE","NETCHANGE","PERCENTCHANGE","MARKETCAP","COUNTRY","IPOYEAR","VOL
UME","SECTOR","INDUSTRY"};
                this.header= table_header;
                header_length = this.header.length;
        }

        @Override
        public void write(List<? extends String> messages) throws Exception {
                long threadId = Thread.currentThread().getId();

                for (String msg : messages) {

                        String[] msg_split = msg.split(",");

                        if(msg_split.length!=this.header_length){
                                System.out.println("Error: "+msg);
                                continue;
                        }
                        StringBuilder msgBuilder= new StringBuilder();

                        for (String s:msg_split){
                                msgBuilder.append(String.format("\"%s\",", s));
                        }
                        msgBuilder.deleteCharAt(msgBuilder.length()-1);

                        String insert_string = String.format("INSERT INTO %s VALUES
(%s)",table_name, msgBuilder.toString());

                        System.out.println(String.format("Thread %d: insert string: %s",
threadId,  insert_string));
                        mysqlJdbcTemplate.batchUpdate(insert_string);
                }
        }

}
```

Parallel processing

Below configuration shows how to use Thread pool to execute a batch job in parallel. It is fairly simple:
hook the chunk step with a ThreadPoolTaskExecutor. Normally we can set the thread pool size to the

number of CPU in the execution machine.   Note that, *SimpleAsyncTaskExecutor* will not resuse threads, it will create a new thread for each chunk process. For more details, refer to Spring web site [6] on Scaling and Parallel Processing.

```
@Configuration
public class ParallelConfig {

……
    @Bean
    public Step readWriteStocksParallel() {
        return stepBuilderFactory.get("readWriterStocksParallel")
                .<String, String> chunk(100)
                .reader(new GenericReader()).processor(new GenericProcessor())
                .writer(new GenericWriter(mysqlJdbcTemplate))
                .taskExecutor(taskExecutor())
                .build();
    }


    @Bean
    public TaskExecutor taskExecutor() {
        ThreadPoolTaskExecutor threadPoolTaskExecutor = new ThreadPoolTaskExecutor();
        threadPoolTaskExecutor.setCorePoolSize(4);
        return threadPoolTaskExecutor;
    }

    //
    // SimpleAsyncTaskExecutor:
    // This starts a new thread and executes it asynchronously.
    // It does not reuse the thread
    //
    @Bean
    public TaskExecutor asyncTaskExecutor() {
        SimpleAsyncTaskExecutor asyncTaskExecutor = new SimpleAsyncTaskExecutor();
        asyncTaskExecutor.setConcurrencyLimit(5);
        return asyncTaskExecutor;
    }
}
```

Console output
With chunk size=10 and pool size=4:

- Reader's before step called once
- 4 threads started, each reads 10 lines from input file.
- Then each thread process 10 line and then write 10 lines.
- Different threads use the same reader, processor, writer objects.

Key points:
Those 4 threads work on the same reader, processor, and writer objects. In another word, those threads work on the same chunk step. Be careful on concurrency issues, need to synchronize those shared objects in reader, processor, writers etc. and database connection pools.

Partition input

The partition input process is complicated, do not recommend this approach unless have to use it.

The key is to use a master step to call the slave steps. The master step split the input into blocks of data and starts the slave steps separately based on the grid size (which can be set to the number of core CPU). The slave step processes each block of data. Therefore, the master step needs to know (1) how to partition the data (need a practitioner), (2) how to execute the process (thread pool executor) (3) the slave step to do the real work. The slave step is the same as previous section chunk step, simply read, process, and write.

The partitioner needs to know the input size to determine each block size. In order to set this value during runtime instead of hardcoded or passed in from a configuration file, I inserted a job listener, to get the total number of lines in the input file, and set it to a shared data structure, in this case if the same config file.

```java
public static int total_lines=0;

@Bean
public Job importStocksPartition() {
    return jobBuilderFactory.get("importStocksPartition")
            .incrementer(new RunIdIncrementer()).listener(new JobListener())
            .flow(masterStep()).end().build();
}

@Bean
public TaskExecutor taskExecutor() {
    ThreadPoolTaskExecutor threadPoolTaskExecutor = new ThreadPoolTaskExecutor();
    threadPoolTaskExecutor.setCorePoolSize(4); // My PC has 4 CPU
    return threadPoolTaskExecutor;
}

@Bean
public InputRangePartitioner partitioner() {
    return new InputRangePartitioner();
}


@Bean
public PartitionHandler partitionHandler() {
    TaskExecutorPartitionHandler taskExecutorPartitionHandler = new
TaskExecutorPartitionHandler();
    //
    // Grid Size:
    // number of data blocks to create to be processed by workers
    // will be passed to partitioner
    //
    taskExecutorPartitionHandler.setGridSize(4);
    taskExecutorPartitionHandler.setTaskExecutor(taskExecutor());
    taskExecutorPartitionHandler.setStep(slaveStep());
    return taskExecutorPartitionHandler;
}

@Bean
public Step slaveStep() {
    //
    // no change on slave step, same as a normal chunk process
    //
    return stepBuilderFactory.get("slaveStep").<String, String>chunk(20)
            .reader(new GenericReader())
            .processor(new GenericProcessor())
            .writer(new GenericWriter(mysqlJdbcTemplate))
            .build();
}

@Bean
public Step masterStep() {
    //
```

```java
        // master step sets slave name, partitioner, partition handler
        //
        return stepBuilderFactory.get("masterStep")
                .partitioner(slaveStep().getName(), partitioner())
                .partitionHandler(partitionHandler())
                .build();
    }


    class InputRangePartitioner implements Partitioner {
        @BeforeStep
        public void beforeStep(StepExecution stepExecution) {

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


            System.out.println("Partitioner gridSize: " + gridSize);
            int min = 1;
            //int max = 100;
            int max = PartitionConfig.total_lines;
            System.out.println("Partitioner max: " + max);
            int targetSize = (max - min) / gridSize + 1;
            System.out.println("targetSize : " + targetSize);
            Map<String, ExecutionContext> result = new HashMap<>();

            int number = 0;
            int start = min;
            int end = start + targetSize - 1;
            //1 to 500
            // 501 to 1000
            while (start <= max) {
                ExecutionContext value = new ExecutionContext();
                result.put("partition" + number, value);

                if (end >= max) {
                    end = max;
                }
                value.putInt("minValue", start);
                value.putInt("maxValue", end);
                start += targetSize;
                end += targetSize;
                number++;
            }
            System.out.println("partition result:" + result);
            return result;
        }
    }


    public class JobListener extends JobExecutionListenerSupport {

        @Override
        public void beforeJob(JobExecution jobExecution) {
            //
            // determine total lines
            //
            JobParameters jobParameters = jobExecution.getJobParameters();
            String fileName = jobParameters.getString("fileName");
            long skip_line = jobParameters.getLong("skip_line");
            int count=0;

            System.out.println("in beforeJob, job param fileName=" + fileName);
            ClassLoader classLoader = getClass().getClassLoader();
            URL resource = classLoader.getResource(fileName);
            File file = new File(resource.getFile());
            System.out.println(file.getPath());
            System.out.printf("File %s exist:%s%n", fileName, file.exists());
            try {
                BufferedReader br = new BufferedReader(new FileReader(file));
```

```java
        for (int i = 0; i < skip_line; i++) {
            System.out.println("Skip:" + br.readLine());
        }
        while (( br.readLine()) != null){
            count++;
        }
        System.out.printf("File %s total lines:%d%n", fileName, count);
        jobExecution.getExecutionContext().putInt("total_lines",count);

        //Partitioner cannot get execution context, need to pass it throw a structure.
        PartitionConfig.total_lines=count;

    } catch (FileNotFoundException e) {
        throw new RuntimeException(e);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

@Override
public void afterJob(JobExecution jobExecution) {
    if (jobExecution.getStatus() == BatchStatus.COMPLETED) {
        System.out.println("BATCH JOB COMPLETED SUCCESSFULLY");
    }
}

}
```

Console output

With chunk size=10 and pool size=4:

- Reader's before step called by each worker (slave) step.
- 4 threads started, each reads 10 lines from input file.
- Then each thread process 10 line and then write 10 lines.
- Different threads use the same reader, processor, writer objects.

Key points:

Partition is typically used for remote processing. Partition master (or manager) step will create multiple identical workers. The number of works are the given grid size. For local processing, those workers are identical, meaning the same reader, writer objects, but execute on different thread with different chunks of input provided by the customized partitioner. However, if the worker step has listeners, before/after step methods, those methods will be call by each worker; on the contrary, those method get called only once in multi-thread step scheme. Other than that, I don't see any differences for local processing.

I personally suggest don't use partition for local processing, use multi-thread step instead. There are so many open source packages, don't use them if you don't feel comfortable.

For more information, please refer to Spring web site [6] on Scaling and Parallel Processing.

Run

Start the spring boot application, use a web browser or a postman, type URL:

1. http://localhost:8080/jobs/importStocks for sequential processing
2. http://localhost:8080/jobs/importStocksParallel for running in parallel
3. http://localhost:8080/jobs/importStocksPartition for partition input

The controller

```java
@RestController
@RequestMapping("/jobs")
public class JobInvokerController {

    @Autowired
    JobLauncher jobLauncher;

    @Autowired
    Job importStocks;

    @Autowired
    Job importStocksParallel;

    @Autowired
    Job importStocksPartition;

    @RequestMapping("/importStocks")
    public String importStocks() throws Exception {

        //
        // hardcode some parameters, can be passed in from url
        //
        String fileName = "input/current_holdings.csv";

        JobParameters jobParameters = new JobParametersBuilder()
                .addLong("time", System.currentTimeMillis())
                .addString("fileName", fileName)
                .toJobParameters();
        jobLauncher.run(importStocks, jobParameters);

        return "Batch job has been invoked";
    }

    /**
     *  load data in multi-threads
     *
     */
    @RequestMapping("/importStocksParallel")
    public String importStocksParallel() throws Exception {
        System.out.println("importStocksParallel...");

        //
        // hardcode some parameters, can be passed in from url
        //
        String fileName = "input/nasdaq_listings.csv";

        JobParameters jobParameters = new JobParametersBuilder()
                .addLong("time", System.currentTimeMillis())
                .addString("fileName", fileName)
                .toJobParameters();
        jobLauncher.run(importStocksParallel, jobParameters);

        return "Batch job has been invoked";
    }

    /**
```

```
 *  load data with multi-thread and input partition
 *
 */
@RequestMapping("/importStocksPartition")
public String importStocksPartition() throws Exception {

    //
    // hardcode some parameters, can be passed in from url
    //
    String fileName = "input/nasdaq_listings.csv";

    JobParameters jobParameters = new JobParametersBuilder()
            .addLong("time", System.currentTimeMillis())
            .addString("fileName", fileName)
            .addLong("skip_line",1L)
            .toJobParameters();
    jobLauncher.run(importStocksPartition, jobParameters);

    return "Batch job has been invoked";
    }
}
```

Fault Tolerance

A fault tolerance job should have skip, retry, restart capabilities, and abnormal behaviors should be traceable.

We only show how to skip, log and redirect the failed records to a file, so it will be easier to correct the failed ones and reload them again.

The Approcah:

- Add a skip listener, redirect errors to a database table
- Add a validation step, compare total read and write record, fail the job if they are not marched. This can be done in job completion listener, but use a tasklet here to demo how to use job execution context to pass parameters between steps.

The skip listener:

```
public class StepSkipListener implements SkipListener<String, String> {
@Override
public void onSkipInRead(Throwable throwable) {
@Override
public void onSkipInWrite(String item, Throwable throwable) {
@Override
public void onSkipInProcess(String item, Throwable throwable) {
        ……
        insertError("writer", msg,item);
        ……
}
```

The chunk step with skip listener:

```
public Step readWriteStocks() {
    return stepBuilderFactory.get("readWriteStocks")
        .<String, String> chunk(1)
```

```
        .reader(new GenericReader())
        .processor(new GenericProcessor())
        .writer(new GenericWriter(mysqlJdbcTemplate))
        .faultTolerant()
        .skipLimit(100)
        .listener(skipListener())
        .skip(Throwable.class) //skip all failures
        .build();
}
```

The job and validation step configuration:

```
@Bean
public Job importStocks() {
    return jobBuilderFactory.get("importStocks")
        .incrementer(new RunIdIncrementer()).listener(listener())
        .flow(readWriteStocks())
        .next(validation())
        .end().build();
}

@Bean
public Step validation() {
    return stepBuilderFactory.get("validation")
        .tasklet(new Validation(mysqlJdbcTemplate))
        .build();
}
```

The validation is a tasklet to verify total read and write count, assembly all needed parameters (such as database tables, read counts etc.) are passed through job execution context.

JUnit Testing

References

1. https://docs.spring.io/spring-batch/docs/current/reference/html/
2. https://www.tutorialspoint.com/spring_batch/spring_batch_basic_application.htm
3. https://mkyong.com/spring/spring-embedded-database-examples/
4. https://github.com/dsong99/Notes/tree/main/Java/simple_batch
5. https://github.com/dsong99/Notes/tree/main/Java/spring_boot_batch
6. https://docs.spring.io/spring-batch/docs/current/reference/html/scalability.html
7.