书接上回: 2024最新!一文看懂Spring Batch批处理(大白话版,干货满满)

八、作业控制

作业的运行指的是对作业的控制，包括作业启动，作业停止，作业异常处理，作业重启处理等。

8.1 作业启动

8.1.1 SpringBoot 启动

目前为止，上面所有的案例都是使用Spring Boot 原生功能来启动作业的，其核心类：**JobLauncherApplicationRunner** ， Spring Boot启动之后，马上调用该类run方法，然后将操作委托给SimpleJobLauncher类run方法执行。默认情况下，Spring Boot一启动马上执行作业。

如果不想Spring Boot启动就执行，可以通过配置进行修改

```
spring:
  batch:
    job:
      enabled: false   #false表示不启动
```

8.1.2 Spring 单元测试启动

开发中如果想简单验证批处理逻辑是否能运行，可以使用单元测试方式启动作业

先引入spring-test测试依赖

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

建立启动类

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

建立测试类

```
package com.langfeiyes.batch._14_job_start_test;

import org.junit.jupiter.api.Test;
import org.springframework.batch.core.*;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
```

```
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.batch.core.scope.context.ChunkContext;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.core.step.tasklet.TaskletStep;
import org.springframework.batch.repeat.RepeatStatus;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest(classes = App.class)
public class StartJobTest {
    //job调度器
    @Autowired
    private JobLauncher jobLauncher;
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;
    public Tasklet tasklet(){
        return new Tasklet() {
            @Override
            public RepeatStatus execute(StepContribution contribution,
ChunkContext chunkContext) throws Exception {
                System.out.println("Hello SpringBatch....");
                return RepeatStatus.FINISHED;
            }
        };
    }
    public Step  step1(){
        TaskletStep step1 = stepBuilderFactory.get("step1")
                .tasklet(tasklet())
                .build();
        return step1;
    }
    //定义作业
    public Job job(){
        Job job = jobBuilderFactory.get("start-test-job")
                .start(step1())
                .build();
        return job;
    }

    @Test
    public void testStart() throws Exception{
        //job作业启动
        //参数1：作业实例，参数2：作业运行携带参数
        jobLauncher.run(job(), new JobParameters());
    }
}
```

跟之前的SpringBoot启动区别在于多了JobLauncher 对象的获取，再由这个对象调用run方法启动。

8.1.3 RESTful API 启动

如果批处理不是SpringBoot启动就启动，而是通过web请求控制，那该怎么办呢？不难，引入web环境即可

1>首先限制，不随SpringBoot启动而启动

```
spring:
  batch:
    job:
      enabled: false   #false表示不启动
```

2>引入web 环境

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

3>编写启动类

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

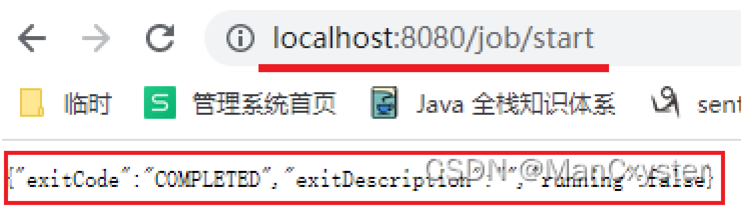
4>编写配置类

```
package com.langfeiyes.batch._15_job_start_restful;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.StepContribution;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.scope.context.ChunkContext;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.core.step.tasklet.TaskletStep;
import org.springframework.batch.repeat.RepeatStatus;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@EnableBatchProcessing
@Configuration
public class BatchConfig {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Bean
    public Tasklet tasklet(){
        return new Tasklet() {
            @Override
            public RepeatStatus execute(StepContribution contribution,
ChunkContext chunkContext) throws Exception {
                System.out.println("Hello SpringBatch....");
                return RepeatStatus.FINISHED;
            }
        };
    }

    @Bean
    public Step step1(){
        TaskletStep step1 = stepBuilderFactory.get("step1")
                .tasklet(tasklet())
                .build();
        return step1;
    }
    //定义作业
    @Bean
    public Job job(){
        Job job = jobBuilderFactory.get("hello-restful-job")
                .start(step1())
                .build();
        return job;
    }
}
```

5>编写Controller类

```
package com.langfeiyes.batch._15_job_start_restful;

import org.springframework.batch.core.*;
import org.springframework.batch.core.launch.JobLauncher;
import
org.springframework.batch.core.repository.JobExecutionAlreadyRunningException;
import
org.springframework.batch.core.repository.JobInstanceAlreadyCompleteException;
import org.springframework.batch.core.repository.JobRestartException;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.Map;
import java.util.Properties;

@RestController
public class HelloController {
    @Autowired
    private JobLauncher jobLauncher;
    @Autowired
    private Job job;
    @GetMapping("/job/start")
    public ExitStatus start() throws Exception {
        //启动job作业
        JobExecution jobExet = launcher.run(job, jp);
        return jobExet.getExitStatus();
    }
}
```

6>测试



localhost:8080/job/start

```
{"exitCode":"COMPLETED","exitDescription":"","running":false}
```

注意：如果需要接收参数

localhost:8080/jo[illegible]

1>作业使用run.id自增

```
//构造一个job对象
@Bean
public Job job(){
    return jobBuilderFactory.get("hello-restful-job")
        .start(step1())
        .incrementer(new RunIdIncrementer())
        .build();
}
```

2>改动HelloController接口方法

```
@RestController
public class HelloController {
    @Autowired
    private JobLauncher launcher;
    @Autowired
    private Job job;
    @Autowired
    private JobExplorer jobExplorer;  //job 展示对象
    @GetMapping("/job/start")
    public ExitStatus startJob(String name) throws Exception {
        //启动job作业
        JobParameters jp = new JobParametersBuilder(jobExplorer)
                .getNextJobParameters(job)
                .addString("name", name)
                .toJobParameters();
        JobExecution jobExet = launcher.run(job, jp);
        return jobExet.getExitStatus();
    }
}
```

8.2 作业停止

作业的停止，存在有3种情况：

- 一种自然结束

  作业成功执行，正常停止，此时作业返回状态为：**COMPLETED**
- 一种异常结束
  作业执行过程因为各种意外导致作业中断而停止，大多数作业返回状态为：**FAILED**
- 一种编程结束

某个步骤处理数据结果不满足下一步骤执行前提，手动让其停止，一般设置返回状态为：**STOPED**

上面1,2种情况相对简单，我们重点说下第三种：以编程方式让作业停止。

模拟一个操作场景

1>有一个资源类，里面有2个属性：总数：totalCount = 100， 读取数：readCount = 0

2>设计2个步骤，step1 用于叠加readCount 模拟从数据库中读取资源， step2 用于执行逻辑

3>当totalCount == readCount 时，为正常情况，正常结束。如果不等时，为异常状态。此时不执行step2，直接停止作业。

4>修复数据，在从step1开始执行，并完成作业

```
public class ResouceCount {
    public static int totalCount = 100;  //总数
    public static int  readCount = 0;    //读取数
}
```

要实现上面需求，有2种方式可以实现

方案1：Step 步骤监听器方式

监听器

```
public class StopStepListener implements StepExecutionListener {
    @Override
    public void beforeStep(StepExecution stepExecution) {
    }

    @Override
    public ExitStatus afterStep(StepExecution stepExecution) {

       //不满足
        if(ResouceCount.totalCount != ResouceCount.readCount){
            return ExitStatus.STOPPED;  //手动停止，后续可以重启
        }
        return stepExecution.getExitStatus();
    }
}
```

代码

```
package com.langfeiyes.batch._16_job_stop;

import com.langfeiyes.batch._01_hello.HelloJob;
```

```java
import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.StepContribution;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.scope.context.ChunkContext;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.repeat.RepeatStatus;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
@EnableBatchProcessing
public class ListenerJobStopJob {

    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    private int readCount = 50; //模拟只读取50个
    @Bean
    public Tasklet tasklet1(){
        return new Tasklet() {
            @Override
            public RepeatStatus execute(StepContribution contribution,
ChunkContext chunkContext) throws Exception {
                for (int i = 1; i <= readCount; i++) {
                    System.out.println("---------------step1执行-"+i+"-----------
-------");
                    ResouceCount.readCount ++;
                }
                return RepeatStatus.FINISHED;
            }
        };
    }

    @Bean
    public Tasklet tasklet2(){
        return new Tasklet() {
            @Override
            public RepeatStatus execute(StepContribution contribution,
ChunkContext chunkContext) throws Exception {
                System.err.println("step2执行了.......");
                System.err.println("readCount:" + ResouceCount.readCount + ",
totalCount:" + ResouceCount.totalCount);
                return RepeatStatus.FINISHED;
            }
        };
    }

    @Bean
    public StopStepListener stopStepListener(){
        return new StopStepListener();
    }
    @Bean
    public Step step1(){
        return stepBuilderFactory.get("step1")
                .tasklet(tasklet1())
                .listener(stopStepListener())
                .allowStartIfComplete(true)  //执行完后，运行重启
                .build();
    }

    @Bean
    public Step step2(){
        return stepBuilderFactory.get("step2")
                .tasklet(tasklet2())
                .build();
    }

    //定义作业
    @Bean
    public Job job(){
        return jobBuilderFactory.get("job-stop-job")
                .start(step1())
                .on("STOPPED").stopAndRestart(step1())
                .from(step1()).on("*").to(step2()).end()
                .build();

    }
    public static void main(String[] args) {
        SpringApplication.run(ListenerJobStopJob.class, args);
    }
}
```

第一次执行：tasklet1 中readCount 默认执行50次，不满足条件， stopStepListener() afterStep 返回STOPPED, job进行条件控制走**.on("STOPPED").stopAndRestart(step1())** 分支，停止并允许重启–下次重启，从step1步骤开始执行

第二次执行， 修改readCount = 100， 再次启动作业，task1遍历100次，满足条件，stopStepListener() afterStep 正常返回，job条件控制走**.from(step1()).on("*").to(step2()).end()**分支，正常结束。

注意：step1() 方法中**.allowStartIfComplete(true)** 代码必须添加，因为第一次执行step1步骤，虽然不满足条件，但是它仍属于正常结束(正常执行完tasklet1的流程)，状态码：COMPLETED， 第二次重启，默认情况下正常结束的step1步骤是不允许再执行的，所以必须设置：**.allowStartIfComplete(true)** 允许step1即使完成也可以重启。

方案2：StepExecution停止标记

```
package com.langfeiyes.batch._17_job_stop_sign;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.StepContribution;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
```

```
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.scope.context.ChunkContext;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.repeat.RepeatStatus;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
@EnableBatchProcessing
public class SignJobStopJob {

    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    private int readCount = 50; //模拟只读取50个
    @Bean
    public Tasklet tasklet1(){
        return new Tasklet() {
            @Override
            public RepeatStatus execute(StepContribution contribution,
ChunkContext chunkContext) throws Exception {
                for (int i = 1; i <= readCount; i++) {
                    System.out.println("---------------step1执行-"+i+"-----------
-------");

                    ResouceCount.readCount ++;
                }

                if(ResouceCount.readCount != ResouceCount.totalCount){

 chunkContext.getStepContext().getStepExecution().setTerminateOnly();
                }

                return RepeatStatus.FINISHED;
            }
        };
    }

    @Bean
    public Tasklet tasklet2(){
        return new Tasklet() {
            @Override
            public RepeatStatus execute(StepContribution contribution,
ChunkContext chunkContext) throws Exception {
                System.err.println("step2执行了.......");
                System.err.println("readCount:" + ResouceCount.readCount + ",
totalCount:" + ResouceCount.totalCount);
                return RepeatStatus.FINISHED;
            }
        };
    }
    @Bean
    public Step step1(){
        return stepBuilderFactory.get("step1")
                .tasklet(tasklet1())
                .allowStartIfComplete(true)
                .build();
    }

    @Bean
    public Step step2(){
        return stepBuilderFactory.get("step2")
                .tasklet(tasklet2())
                .build();
    }

    //定义作业
    @Bean
    public Job job(){
         return jobBuilderFactory.get("job-stop-job")
                .start(step1())
                //.on("STOPPED").stopAndRestart(step1())
                //.from(step1()).on("*").to(step2()).end()
                .next(step2())
                .build();

    }
    public static void main(String[] args) {
        SpringApplication.run(SignJobStopJob.class, args);
    }
}
```

变动的代码有2处

tasket1(), 多了下面判断

```
if(ResouceCount.readCount != ResouceCount.totalCount){
    chunkContext.getStepContext().getStepExecution().setTerminateOnly();
}
```

其中的StepExecution#setTerminateOnly() 给运行中的stepExecution设置停止标记，Spring Batch 识别后直接停止步骤，进而停止流程

job() 改动

```
return jobBuilderFactory.get("job-stop-job")
    .start(step1())
    .next(step2())
    .build();
```

正常设置步骤流程。

8.3 作业重启

作业重启，表示允许作业步骤重新执行，默认情况下，只允许异常或终止状态的步骤重启，但有时存在特殊场景，要求需要其他状态步骤重启，为应付各种复杂的情形，Spring Batch 提供3种重启控制操作。

8.3.1 禁止重启

这种适用一次性执行场景，如果执行失败，就不允许再次执行。可以使用作业的禁止重启逻辑

```
package com.langfeiyes.batch._18_job_restart_forbid;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.StepContribution;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.scope.context.ChunkContext;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.repeat.RepeatStatus;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
@EnableBatchProcessing
public class JobForBidRestartJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Bean
    public Tasklet tasklet1(){
        return new Tasklet() {
            @Override
            public RepeatStatus execute(StepContribution contribution,
ChunkContext chunkContext) throws Exception {
                System.err.println("-------------tasklet1-------------");


 chunkContext.getStepContext().getStepExecution().setTerminateOnly(); //停止步骤
                return RepeatStatus.FINISHED;

            }
        };
    }

    @Bean
    public Tasklet tasklet2(){
        return new Tasklet() {
            @Override
```

```java
            public RepeatStatus execute(StepContribution contribution,
ChunkContext chunkContext) throws Exception {
                System.err.println("------------tasklet2-------------");
                return RepeatStatus.FINISHED;
            }
        };
    }


    @Bean
    public Step step1(){
        return stepBuilderFactory.get("step1")
                .tasklet(tasklet1())
                .build();
    }

    @Bean
    public Step step2(){
        return stepBuilderFactory.get("step2")
                .tasklet(tasklet2())
                .build();
    }

    //定义作业
    @Bean
    public Job job(){
        return jobBuilderFactory.get("job-forbid-restart-job")
                .preventRestart()  //禁止重启
                .start(step1())
                .next(step2())
                .build();

    }
    public static void main(String[] args) {
        SpringApplication.run(JobForBidRestartJob.class, args);
    }
}
```

观察上面代码，比较特别之处：

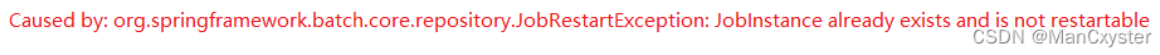
tasklet1() 加了setTerminateOnly 设置，表示让步骤退出

```
chunkContext.getStepContext().getStepExecution().setTerminateOnly();
```

job() 多了**.preventRestart()** 逻辑，表示步骤不允许重启

第一次按上面的代码执行一次， step1() 状态为 **STOPPED**

第二次去掉setTerminateOnly逻辑，重新启动步骤，观察结果，直接报错



Caused by: org.springframework.batch.core.repository.JobRestartException: JobInstance already exists and is not restartable

8.3.2 限制重启次数

适用于重启次数有限的场景，比如下载/读取操作，可能因为网络原因导致下载/读取失败，运行重试几次，但是不能无限重试。这时可以对步骤执行进行重启次数限制。

```
package com.langfeiyes.batch._19_job_restart_limit;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.StepContribution;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.scope.context.ChunkContext;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.repeat.RepeatStatus;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
@EnableBatchProcessing
public class JobLimitRestartJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Bean
    public Tasklet tasklet1(){
        return new Tasklet() {
            @Override
            public RepeatStatus execute(StepContribution contribution,
ChunkContext chunkContext) throws Exception {
                System.err.println("-------------tasklet1-------------");


 chunkContext.getStepContext().getStepExecution().setTerminateOnly(); //停止步骤
                return RepeatStatus.FINISHED;

            }
        };
    }

    @Bean
    public Tasklet tasklet2(){
        return new Tasklet() {
            @Override
            public RepeatStatus execute(StepContribution contribution,
ChunkContext chunkContext) throws Exception {
                System.err.println("-------------tasklet2-------------");
                return RepeatStatus.FINISHED;
            }
        };
    }


    @Bean
    public Step step1(){
        return stepBuilderFactory.get("step1")
                .startLimit(2)
                .tasklet(tasklet1())
                .build();
    }

    @Bean
    public Step step2(){
        return stepBuilderFactory.get("step2")
                .tasklet(tasklet2())
                .build();
    }

    //定义作业
    @Bean
    public Job job(){
        return jobBuilderFactory.get("job-restart-limit-job")
                .start(step1())
                .next(step2())
                .build();

    }
    public static void main(String[] args) {
        SpringApplication.run(JobLimitRestartJob.class, args);
    }
}
```

变动:

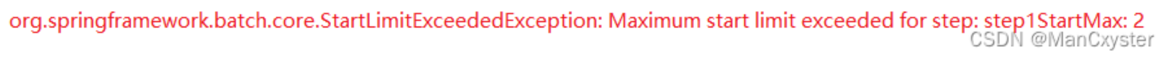
step1() 添加了**.startLimit(2)** 表示运行重启2次，注意，第一次启动也算一次

tasklet1() 设置setTerminateOnly 第一次先让step1 状态为**STOPPED**

第一次执行， step1 为 **STOPPED** 状态

第二次执行，不做任何操作，第二次执行，step1 还是STOPPED状态

第三次执行， 注释掉tasklet1() 中setTerminateOnly， 查询结果


org.springframework.batch.core.StartLimitExceededException: Maximum start limit exceeded for step: step1StartMax: 2

8.3.3 无限重启

Spring Batch 限制同job名跟同标识参数作业只能成功执行一次，这是Spring Batch 定理，无法改变的。但是，对于步骤不一定适用，可以通过步骤的allowStartIfComplete(true) 实现步骤的无限重启。

```
package com.langfeiyes.batch._20_job_restart_allow;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import org.springframework.batch.core.StepContribution;
```

```
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.scope.context.ChunkContext;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.repeat.RepeatStatus;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
@EnableBatchProcessing
public class JobAllowRestartJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Bean
    public Tasklet tasklet1(){
        return new Tasklet() {
            @Override
            public RepeatStatus execute(StepContribution contribution,
ChunkContext chunkContext) throws Exception {
                System.err.println("-------------tasklet1-------------");
                return RepeatStatus.FINISHED;

            }
        };
    }

    @Bean
    public Tasklet tasklet2(){
        return new Tasklet() {
            @Override
            public RepeatStatus execute(StepContribution contribution,
ChunkContext chunkContext) throws Exception {
                System.err.println("-------------tasklet2-------------");
                return RepeatStatus.FINISHED;
            }
        };
    }


    @Bean
    public Step step1(){
        return stepBuilderFactory.get("step1")
                .tasklet(tasklet1())
                .build();
    }

    @Bean
    public Step step2(){
        return stepBuilderFactory.get("step2")
```

```
                .tasklet(tasklet2())
                .build();
    }

    //定义作业
    @Bean
    public Job job(){
        return jobBuilderFactory.get("job-allow-restart-job")
                .start(step1())
                .next(step2())
                .build();

    }
    public static void main(String[] args) {
        SpringApplication.run(JobAllowRestartJob.class, args);
    }
}
```

观察上面代码，很正常逻辑

第一次启动：step1 step2正常执行，整个Job 成功执行完成

第二次启动：不做任何改动时，再次启动job，没有报错，但是观察数据库表batch_job_execution 状态为 **NOOP** 无效执行，step1 step2 不会执行。

第三次启动：给step1 step2 添加上**.allowStartIfComplete(true)**， 再次启动，一切正常，并且可以无限启动

九、ItemReader

居于块操作的步骤由一个ItemReader，一个ItemProcessor和一个ItemWriter组成，一个负责读取数据，一个负责处理数据，一个负责输出数据，上一章节讲完步骤，接下来就重点讲解Spring Batch 输入组件：**ItemReader**

ItemReader 是Spring Batch 提供的输入组件，规范接口是ItemReader, 里面有个read() 方法，我们可以实现该接口去定制输入逻辑。

```
public interface ItemReader<T> {
    @Nullable
    T read() throws Exception, UnexpectedInputException, ParseException,
NonTransientResourceException;
}

```

Spring Batch 根据常用的输入类型，提供许多默认的实现，包括：平面文件、数据库、JMS资源和其他输入源等，接下来一起操作一下比较场景的输入场景。

9.1 读平面文件

平面文件一般指的都是简单行/多行结构的纯文本文件，比如记事本记录文件。与xml这种区别在于没有结构，没有标签的限制。Spring Batch默认使用 FlatFileItemReader 实现平面文件的输入。

9.1.1 方式1：delimited–字符串截取

需求：读取user.txt文件，解析出所有用户信息

user.txt

```
1#dafei#18
2#xiaofei#16
3#laofei#20
4#zhongfei#19
5#feifei#15
```

实体类

```java
@Getter
@Setter
@ToString
public class User {
    private Long id;
    private String name;
    private int age;
}
```

实现作业

```java
package com.langfeiyes.batch._21_itemreader_flat;

import com.langfeiyes.batch._20_job_restart_allow.JobAllowRestartJob;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;

import java.util.List;

@SpringBootApplication
@EnableBatchProcessing
public class FlatReaderJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;
    @Bean
    public FlatFileItemReader<User> userItemReader(){
        return new FlatFileItemReaderBuilder<User>()
                .name("userItemReader")
                .resource(new ClassPathResource("users.txt"))
                .delimited().delimiter("#")
                .names("id", "name", "age")
                .targetType(User.class)

                .build();
    }

    @Bean
    public ItemWriter<User> itemWriter(){
        return new ItemWriter<User>() {
            @Override
            public void write(List<? extends User> items) throws Exception {
                items.forEach(System.err::println);
            }
        };
    }

    @Bean
    public Step step(){
        return stepBuilderFactory.get("step1")
                .<User, User>chunk(1)
                .reader(userItemReader())
                .writer(itemWriter())
                .build();

    }

    @Bean
    public Job job(){
        return jobBuilderFactory.get("flat-reader-job")
                .start(step())
                .build();
    }
    public static void main(String[] args) {
        SpringApplication.run(FlatReaderJob.class, args);
    }
}
```

核心在userItemReader() 实例方法

```
//FlatFileItemReader spring batch 平面文件读入类
//这个类操作特点：一行一行的读数据
@Bean
public FlatFileItemReader<User> userItemReader(){
    return new FlatFileItemReaderBuilder<User>()
        .name("userItemReader")
        .resource(new ClassPathResource("users.txt"))  //指定读取的文件
        .delimited().delimiter("#")  //读取出一行数据，该如何分割数据，默认以,分割，当前使用#号分割
        .targetType(User.class)      //读取出一行数据封装成什么对象
        //给分割后数据打name标记，后续跟User对象属性进行映射
        .names("id", "name", "age")
        .build();
}
```

除了上面讲到的核心方法，FlatFileItemReaderBuilder还提供.**fieldSetMapper .lineTokenizer** 2个方法，用于定制文件解析与数据映射。

9.1.2 方式2：FieldSetMapper-字段映射

FlatFileItemReaderBuilder 提供的方法，用于字段映射，方法参数是一个FieldSetMapper接口对象

```
public interface FieldSetMapper<T> {
    T mapFieldSet(FieldSet fieldSet) throws BindException;
}
```

FieldSet 字段集合，FlatFileItemReader 解析出一行数据，会将这行数据封装到FieldSet对象中。

我们用一个案例来解释一下FieldSetMapper 用法

编写users2.txt文件

```
1#dafei#18#广东#广州#天河区
2#xiaofei#16#四川#成都#武侯区
3#laofei#20#广西#桂林#雁山区
4#zhongfei#19#广东#广州#白云区
5#feifei#15#广东#广州#越秀区
```

用户对象

```
@Getter
@Setter
@ToString
public class User {
    private Long id;
    private String name;
    private int age;
    private String address;
}
```

观察，user2.txt文件中有 id name age province city area 按理用户对象属性应该一一对应，但是此时User只有address，也就是说，后续要将 province ， city ， area 合并成 address 地址值。此时怎么办？这是就需要自定义FieldSetMapper 啦。

```
public class UserFieldMapper implements FieldSetMapper<User> {
    @Override
    public User mapFieldSet(FieldSet fieldSet) throws BindException {

        //自己定义映射逻辑
        User User = new User();
        User.setId(fieldSet.readLong("id"));
        User.setAge(fieldSet.readInt("age"));
        User.setName(fieldSet.readString("name"));
        String addr = fieldSet.readString("province") + " "
                + fieldSet.readString("city") + " " +
fieldSet.readString("area");
        User.setAddress(addr);
        return User;
    }
}
```

上面代码实现FieldSet与User对象映射，将province city area 合并成一个属性address。另外readXxx是FieldSet 独有的方法，Xxx是java基本类型。

```
package com.langfeiyes.batch._22_itemreader_flat_mapper;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;

import java.util.List;

@SpringBootApplication
@EnableBatchProcessing
public class MapperFlatReaderJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Bean
    public UserFieldMapper userFieldMapper(){
        return new UserFieldMapper();
    }


    @Bean
    public FlatFileItemReader<User> userItemReader(){
        return new FlatFileItemReaderBuilder<User>()
```

```
                .name("userMapperItemReader")
                .resource(new ClassPathResource("users2.txt"))
                .delimited().delimiter("#")
                .names("id", "name", "age", "province", "city", "area")
                .fieldSetMapper(userFieldMapper())
                .build();
    }

    @Bean
    public ItemWriter<User> itemWriter(){
        return new ItemWriter<User>() {
            @Override
            public void write(List<? extends User> items) throws Exception {
                items.forEach(System.err::println);
            }
        };
    }

    @Bean
    public Step step(){
        return stepBuilderFactory.get("step1")
                .<User, User>chunk(1)
                .reader(userItemReader())
                .writer(itemWriter())
                .build();

    }

    @Bean
    public Job job(){
        return jobBuilderFactory.get("mapper-flat-reader-job")
                .start(step())
                .build();
    }
    public static void main(String[] args) {
        SpringApplication.run(MapperFlatReaderJob.class, args);
    }
}
```

上面代码核心在userItemReader实例方法

.fieldSetMapper(userFieldMapper())：用上自定义的字段映射器

.names(“id”, “name”, “age”, “province”, “city”, “area”)： users2.txt 每一行使用#分割出现6列，给每一列取名字，然后将其封装到FieldSet对象中

.targetType(User.class)： 注意，使用了fieldSetMapper 之后，不需要在加上这行

9.2 读JSON文件

Spring Batch 也提供专门操作Json文档的API： JsonItemReader，具体使用且看案例

需求：读取下面json格式文档

```
[
  {"id":1, "name":"dafei", "age":18},
  {"id":2, "name":"xiaofei", "age":17},
  {"id":3, "name":"zhongfei", "age":16},
  {"id":4, "name":"laofei", "age":15},
  {"id":5, "name":"feifei", "age":14}
]
```

封装成User对象

```
@Getter
@Setter
@ToString
public class User {
    private Long id;
    private String name;
    private int age;
}
```

```
package com.langfeiyes.batch._23_itemreader_flat_json;

import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.json.JacksonJsonObjectReader;
import org.springframework.batch.item.json.JsonItemReader;
import org.springframework.batch.item.json.builder.JsonItemReaderBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;

import java.util.List;

@SpringBootApplication
@EnableBatchProcessing
public class JsonFlatReaderJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;



    @Bean
    public JsonItemReader<User> userItemReader(){

        ObjectMapper objectMapper = new ObjectMapper();
        JacksonJsonObjectReader<User> jsonObjectReader = new
JacksonJsonObjectReader<>(User.class);
        jsonObjectReader.setMapper(objectMapper);

        return new JsonItemReaderBuilder<User>()
                .name("userJsonItemReader")
                .jsonObjectReader(jsonObjectReader)
                .resource(new ClassPathResource("users.json"))
            .build();
    }

    @Bean
    public ItemWriter<User> itemWriter(){
        return new ItemWriter<User>() {
            @Override
            public void write(List<? extends User> items) throws Exception {
                items.forEach(System.err::println);
            }
        };
    }

    @Bean
    public Step step(){
        return stepBuilderFactory.get("step1")
                .<User, User>chunk(1)
                .reader(userItemReader())
                .writer(itemWriter())
                .build();

    }

    @Bean
    public Job job(){
        return jobBuilderFactory.get("json-flat-reader-job")
                .start(step())
                .build();
    }
    public static void main(String[] args) {
        SpringApplication.run(JsonFlatReaderJob.class, args);
    }
}
```

上面代码核心在：userItemReader() 实例方法，明确指定转换成json格式需要使用转换器，本次使用的Jackson

9.3 读数据库

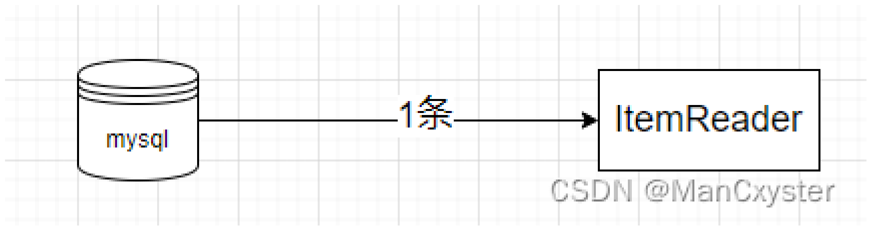
下面是一张用户表user， 如果数据是存放在数据库中，那么又该怎么读取？

```
CREATE TABLE `user` (
  `id` bigint NOT NULL AUTO_INCREMENT COMMENT '主键',
  `name` varchar(255) DEFAULT NULL COMMENT '用户名',
  `age` int DEFAULT NULL COMMENT '年龄',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8mb3;
```

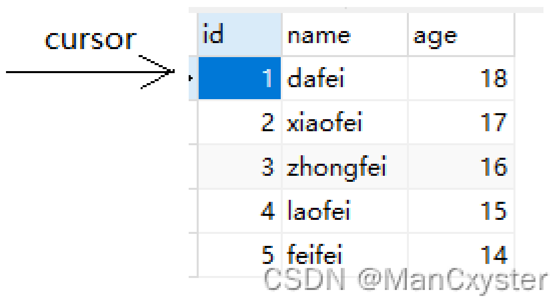
```sql
INSERT INTO `user` VALUES (1, 'dafei', 18);
INSERT INTO `user` VALUES (2, 'xiaofei', 17);
INSERT INTO `user` VALUES (3, 'zhongfei', 16);
INSERT INTO `user` VALUES (4, 'laofei', 15);
INSERT INTO `user` VALUES (5, 'feifei', 14);
```

Spring Batch 提供2种从数据库中读取数据的方式：

9.3.1 居于游标方式



游标是数据库中概念，可以简单理解为一个指针



id	name	age
1	dafei	18
2	xiaofei	17
3	zhongfei	16
4	laofei	15
5	feifei	14

游标遍历时，获取数据表中某一条数据，如果使用JDBC操作，游标指向的那条数据会被封装到ResultSet中，如果想将数据从ResultSet读取出来，需要借助Spring Batch 提供RowMapper 实现表数据与实体对象的映射。

```
user表数据---->User对象
```

Spring Batch JDBC 实现数据表读取需要做几个准备

1>实体对象User

```java
@Getter
@Setter
@ToString
public class User {
    private Long id;
    private String name;
    private int age;
}
```

2>RowMapper 表与实体对象映射实现类

```java
public class UserRowMapper implements RowMapper<User> {
    @Override
    public User mapRow(ResultSet rs, int rowNum) throws SQLException {
        User user = new User();
        user.setId(rs.getLong("id"));
        user.setName(rs.getString("name"));
        user.setAge(rs.getInt("age"));
        return user;
    }
}
```

3>JdbcCursorItemReader编写

```java
package com.langfeiyes.batch._24_itemreader_db_cursor;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.database.JdbcCursorItemReader;
import
org.springframework.batch.item.database.builder.JdbcCursorItemReaderBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

import javax.sql.DataSource;
import java.util.List;

@SpringBootApplication
@EnableBatchProcessing
public class CursorDBReaderJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Autowired
    private DataSource dataSource;

    @Bean
    public UserRowMapper userRowMapper(){
        return new UserRowMapper();
    }

    @Bean
    public JdbcCursorItemReader<User> userItemReader(){

        return new JdbcCursorItemReaderBuilder<User>()
                .name("userCursorItemReader")
                .dataSource(dataSource)
                .sql("select * from user")
                .rowMapper(userRowMapper())
                .build();
    }

    @Bean
    public ItemWriter<User> itemWriter(){
        return new ItemWriter<User>() {
            @Override
            public void write(List<? extends User> items) throws Exception {
                items.forEach(System.err::println);
            }
        };
    }

    @Bean
    public Step step(){
        return stepBuilderFactory.get("step1")
                .<User, User>chunk(1)
                .reader(userItemReader())
                .writer(itemWriter())
                .build();

    }

    @Bean
    public Job job(){
        return jobBuilderFactory.get("cursor-db-reader-job")
                .start(step())
                .build();
    }
    public static void main(String[] args) {
        SpringApplication.run(CursorDBReaderJob.class, args);
    }
}
```

解析：

1>操作数据库，需要引入DataSource

2>留意userItemReader() 方法，需要明确指定操作数据库sql

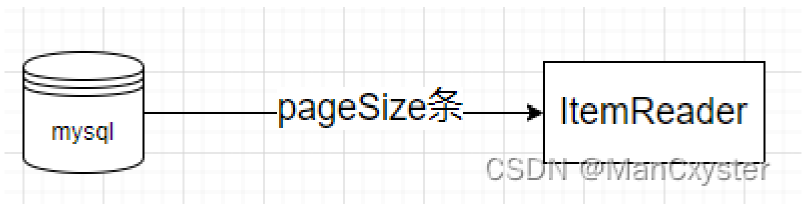
3>留意userItemReader() 方法，需要明确指定游标回来之后，数据映射规则：rowMapper

这里要注意，如果sql需要where 条件，需要额外定义

比如： 查询 age > 16的用户

```
@Bean
public JdbcCursorItemReader<User> userItemReader(){

    return new JdbcCursorItemReaderBuilder<User>()
        .name("userCursorItemReader")
        .dataSource(dataSource)
        .sql("select * from user where age > ?")
        .rowMapper(userRowMapper())
        //拼接参数
        .preparedStatementSetter(new ArgumentPreparedStatementSetter(new
Object[]{16}))
        .build();
}
```

9.3.2 居于分页方式



游标的方式是查询出所有满足条件的数据，然后一条一条读取，而分页是按照指定设置的pageSize数，一次性读取pageSize条。

分页查询方式需要几个要素

1>实体对象，跟游标方式一样

2>RowMapper映射对象，跟游标方式一样

3>数据源，跟游标方式一样

4>PagingQueryProvider 分页逻辑提供者

```
package com.langfeiyes.batch._25_itemreader_db_page;


import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.database.JdbcCursorItemReader;
import org.springframework.batch.item.database.JdbcPagingItemReader;
import org.springframework.batch.item.database.PagingQueryProvider;
import
org.springframework.batch.item.database.builder.JdbcCursorItemReaderBuilder;
import
org.springframework.batch.item.database.builder.JdbcPagingItemReaderBuilder;
import
org.springframework.batch.item.database.support.SqlPagingQueryProviderFactoryBea
n;
```

```
import
org.springframework.batch.item.database.support.SqlitePagingQueryProvider;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.jdbc.core.ArgumentPreparedStatementSetter;

import javax.sql.DataSource;
import java.util.HashMap;
import java.util.List;

@SpringBootApplication
@EnableBatchProcessing
public class PageDBReaderJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Autowired
    private DataSource dataSource;

    @Bean
    public UserRowMapper userRowMapper(){
        return new UserRowMapper();
    }


    @Bean
    public PagingQueryProvider pagingQueryProvider() throws Exception {
        SqlPagingQueryProviderFactoryBean factoryBean = new
SqlPagingQueryProviderFactoryBean();
        factoryBean.setDataSource(dataSource);
        factoryBean.setSelectClause("select *");   //查询列
        factoryBean.setFromClause("from user");    //查询的表
        factoryBean.setWhereClause("where age > :age"); //where 条件
        factoryBean.setSortKey("id");   //结果排序
        return factoryBean.getObject();
    }

    @Bean
    public JdbcPagingItemReader<User> userItemReader() throws Exception {
        HashMap<String, Object> param = new HashMap<>();
        param.put("age", 16);
        return new JdbcPagingItemReaderBuilder<User>()
                .name("userPagingItemReader")
                .dataSource(dataSource)  //数据源
                .queryProvider(pagingQueryProvider())  //分页逻辑
                .parameterValues(param)   //条件
                .pageSize(10) //每页显示条数
                .rowMapper(userRowMapper())  //映射规则
                .build();
    }

    @Bean
    public ItemWriter<User> itemWriter(){
        return new ItemWriter<User>() {
            @Override
            public void write(List<? extends User> items) throws Exception {
                items.forEach(System.err::println);
            }
        };
    }

    @Bean
    public Step step() throws Exception {
        return stepBuilderFactory.get("step1")
                .<User, User>chunk(1)
                .reader(userItemReader())
                .writer(itemWriter())
                .build();

    }

    @Bean
    public Job job() throws Exception {
        return jobBuilderFactory.get("page-db-reader-job1")
                .start(step())
                .build();
    }
    public static void main(String[] args) {
        SpringApplication.run(PageDBReaderJob.class, args);
    }
}
```

解析：

1>需要提供pagingQueryProvider 用于拼接分页SQL

2>userItemReader() 组装分页查询逻辑。

9.4 读取异常

任何输入都有可能存在异常情况，那Spring Batch 如何应对输入异常情况呢？ 3种操作逻辑：

1>跳过异常记录

这里逻辑是当Spring Batch 在读取数据时，根据各种意外情况抛出不同异常，ItemReader 可以按照约定跳过指定的异常，同时也可以限制跳过次数。

```
@Bean
public Step step() throws Exception {
    return stepBuilderFactory.get("step1")
        .<User, User>chunk(1)
        .reader(userItemReader())
        .writer(itemWriter())
        .faultTolerant() //容错
        .skip(Exception.class)  //跳过啥异常
        .noSkip(RuntimeException.class)  //不能跳过啥异常
        .skipLimit(10)  //跳过异常次数
        .skipPolicy(new SkipPolicy() {
            @Override
            public boolean shouldSkip(Throwable t, int skipCount) throws
SkipLimitExceededException {
```

```
                    //定制跳过异常与异常次数
                    return false;
                }
            })
            .build();

}
```

如果出错直接跳过去，这操作有点自欺欺人，并不是优雅的解决方案。开发可选下面这种。

2>异常记录记日志

所谓记录日志，就是当ItemReader 读取数据抛出异常时，将具体数据信息记录下来，方便后续人工接入。

具体实现使用ItemReader监听器。

```
public class ErrorItemReaderListener implements ItemReadListener {
    @Override
    public void beforeRead() {

    }

    @Override
    public void afterRead(Object item) {

    }

    @Override
    public void onReadError(Exception ex) {
        System.out.println("记录读数据相关信息...");
    }
}
```

3>放弃处理

这种异常在处理不是很重要数据时候使用。

十、ItemProcessor

前面我们多次讲过，居于块的读与写，中间还夹着一个ItemProcessor 条目处理。当我们通过ItemReader 将数据读取出来之后，你面临2个选择：

1>直接将数据转向输出

2>对读入的数据进行再加工。

如果选择第一种，那ItemProcessor 可以不用出现，如果选择第二种，就需要引入ItemProcessor 条目处理组件啦。

Spring Batch 为Processor 提供默认的处理器与自定义处理器2种模式以满足各种需求。

10.1 默认ItemProcessor

Spring Batch 提供现成的ItemProcessor 组件有4种：

10.1.1 ValidatingItemProcessor：校验处理器

这个好理解，很多时候ItemReader读出来的数据是相对原始的数据，并没有做过多的校验

数据文件users-validate.txt

```
1##18
2##16
3#laofei#20
4#zhongfei#19
5#feifei#15
```

比如上面文本数据，第一条，第二条name数值没有指定，在ItemReader 读取之后，必定将 "" 空串封装到User name属性中，语法上没有错，但逻辑上可以做文章，比如：用户名不为空。

解决上述问题，可以使用Spring Batch 提供ValidatingItemProcessor 校验器处理。

接下来我们看下ValidatingItemProcessor 怎么实现

1>导入校验依赖

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

2>定义实体对象

```
@Getter
@Setter
@ToString
public class User {
    private Long id;
    @NotBlank(message = "用户名不能为null或空串")
    private String name;
    private int age;
}
```

3>实现

```
package com.langfeiyes.batch._26_itemprocessor_validate;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.file.FlatFileItemReader;
```

```java
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.batch.item.validator.BeanValidatingItemProcessor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;
import org.springframework.util.StringUtils;

import java.util.List;

@SpringBootApplication
@EnableBatchProcessing
public class ValidationProcessorJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;


    @Bean
    public FlatFileItemReader<User> userItemReader(){
        return new FlatFileItemReaderBuilder<User>()
                .name("userItemReader")
                .resource(new ClassPathResource("users-validate.txt"))
                .delimited().delimiter("#")
                .names("id", "name", "age")
                .targetType(User.class)
                .build();
    }

    @Bean
    public ItemWriter<User> itemWriter(){
        return new ItemWriter<User>() {
            @Override
            public void write(List<? extends User> items) throws Exception {
                items.forEach(System.err::println);
            }
        };
    }


    @Bean
    public BeanValidatingItemProcessor<User> beanValidatingItemProcessor(){
        BeanValidatingItemProcessor<User> beanValidatingItemProcessor = new
BeanValidatingItemProcessor<>();
        beanValidatingItemProcessor.setFilter(true);  //不满足条件丢弃数据

        return beanValidatingItemProcessor;
    }


    @Bean
    public Step step(){
        return stepBuilderFactory.get("step1")
                .<User, User>chunk(1)
                .reader(userItemReader())
                .processor(beanValidatingItemProcessor())
                .writer(itemWriter())
                .build();

    }

    @Bean
    public Job job(){
        return jobBuilderFactory.get("validate-processor-job4")
                .start(step())
                .build();
    }
    public static void main(String[] args) {
        SpringApplication.run(ValidationProcessorJob.class, args);
    }
}
```

解析：

1>核心是beanValidatingItemProcessor() 实例方法，核心BeanValidatingItemProcessor 类是Spring Batch 提供现成的Validator校验类，这里直接使用即可。BeanValidatingItemProcessor 是 ValidatingItemProcessor 子类

2> step()实例方法，多了**.processor(beanValidatingItemProcessor())** 操作，引入ItemProcessor组件。

10.1.2 ItemProcessorAdapter：适配器处理器

开发中，很多的校验逻辑已经有现成的啦，那做ItemProcessor处理时候，是否能使用现成逻辑呢？答案 是：yes

比如：现有处理逻辑：将User对象中name转换成大写

```
public class UserServiceImpl{
    public User toUppeCase(User user){
        user.setName(user.getName().toUpperCase());
        return user;
    }
}
```

新建users-adapter.txt 文件，用于测试

```
1#dafei#18
2#xiaofei#16
3#laofei#20
4#zhongfei#19
5#feifei#15
```

完整的逻辑

```
package com.langfeiyes.batch._27_itemprocessor_adapter;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
```

```java
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.adapter.ItemProcessorAdapter;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;

import java.util.List;

@SpringBootApplication
@EnableBatchProcessing
public class AdapterProcessorJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;


    @Bean
    public FlatFileItemReader<User> userItemReader(){
        return new FlatFileItemReaderBuilder<User>()
                .name("userItemReader")
                .resource(new ClassPathResource("users-adapter.txt"))
                .delimited().delimiter("#")
                .names("id", "name", "age")
                .targetType(User.class)
                .build();
    }

    @Bean
    public ItemWriter<User> itemWriter(){
        return new ItemWriter<User>() {
            @Override
            public void write(List<? extends User> items) throws Exception {
                items.forEach(System.err::println);
            }
        };
    }
    @Bean
    public UserServiceImpl userService(){
        return new UserServiceImpl();
    }
    @Bean
    public ItemProcessorAdapter<User, User> itemProcessorAdapter(){
        ItemProcessorAdapter<User, User> adapter = new ItemProcessorAdapter<>();
        adapter.setTargetObject(userService());
        adapter.setTargetMethod("toUppeCase");

        return adapter;
    }
    @Bean
    public Step step(){
        return stepBuilderFactory.get("step1")
                .<User, User>chunk(1)
                .reader(userItemReader())
                .processor(itemProcessorAdapter())
                .writer(itemWriter())
                .build();

    }
    @Bean
    public Job job(){
        return jobBuilderFactory.get("adapter-processor-job")
                .start(step())
                .build();
    }
    public static void main(String[] args) {
        SpringApplication.run(AdapterProcessorJob.class, args);
    }
}
```

解析：

观察itemProcessorAdapter()实例方法，引入ItemProcessorAdapter 适配器类，绑定自定义的UserServiceImpl 类与toUppeCase方法，当ItemReader 读完之后，马上调用UserServiceImpl 类的toUppeCase 方法处理逻辑。方法传参数会被忽略，ItemProcessor会自动处理。

10.1.3 ScriptItemProcessor：脚本处理器

前面要实现User name 变大写，需要大费周折，又定义类，又是定义方法，能不能简化一点。答案也是yes， Spring Batch 提供js脚本的形式，将上面逻辑写到js文件中，加载这文件，就可以实现，省去定义类，定义方法的麻烦。

需求：使用js脚本方式实现用户名大写处理

userScript.js

```
item.setName(item.getName().toUpperCase());
item;
```

这里注意：

1>item是约定的单词，表示ItemReader读除来每个条目

2>userScript.js文件放置到resource资源文件中

完整代码

```
package com.langfeiyes.batch._28_itemprocessor_script;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
```

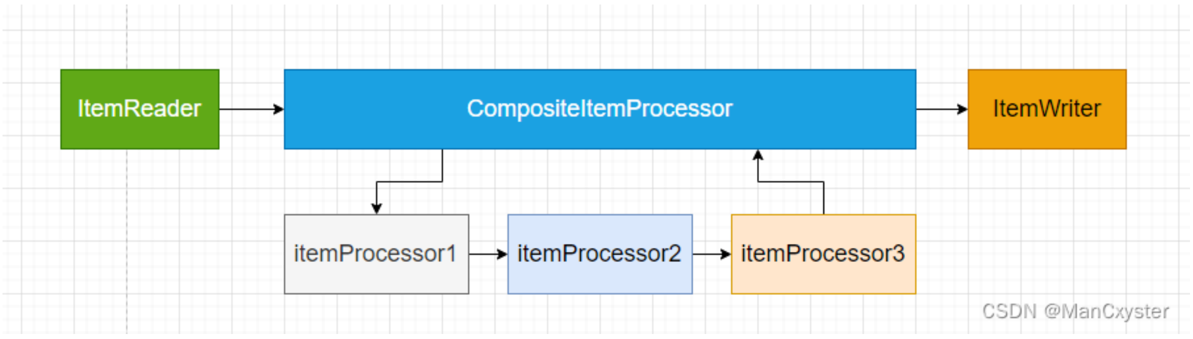
```java
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.batch.item.support.ScriptItemProcessor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;

import java.util.List;

@SpringBootApplication
@EnableBatchProcessing
public class ScriptProcessorJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;


    @Bean
    public FlatFileItemReader<User> userItemReader(){
        return new FlatFileItemReaderBuilder<User>()
                .name("userItemReader")
                .resource(new ClassPathResource("users-adapter.txt"))
                .delimited().delimiter("#")
                .names("id", "name", "age")
                .targetType(User.class)
                .build();
    }

    @Bean
    public ItemWriter<User> itemWriter(){
        return new ItemWriter<User>() {
            @Override
            public void write(List<? extends User> items) throws Exception {
                items.forEach(System.err::println);
            }
        };
    }


    @Bean
    public ScriptItemProcessor<User, User> scriptItemProcessor(){
        ScriptItemProcessor<User, User> scriptItemProcessor = new
ScriptItemProcessor();
        scriptItemProcessor.setScript(new ClassPathResource("userScript.js"));
        return scriptItemProcessor;
    }

    @Bean
    public Step step(){
        return stepBuilderFactory.get("step1")
                .<User, User>chunk(1)
                .reader(userItemReader())
                .processor(scriptItemProcessor())
                .writer(itemWriter())
                .build();

    }

    @Bean
    public Job job(){
        return jobBuilderFactory.get("script-processor-job")
                .start(step())
                .build();
    }
    public static void main(String[] args) {
        SpringApplication.run(ScriptProcessorJob.class, args);
    }
}
```

解析：

核心还是scriptItemProcessor() 实例方法，ScriptItemProcessor 类用于加载js 脚本并处理js脚本。

10.1.4 CompositeItemProcessor：组合处理器

CompositeItemProcessor是一个ItemProcessor处理组合，类似于过滤器链，数据先经过第一个处理器，然后再经过第二个处理器，直到最后。前一个处理器处理的结果，是后一个处理器的输出。



需求：将解析出来用户name进行判空处理，并将name属性转换成大写

1>读取文件：users-validate.txt

```
1##18
2##16
3#laofei#20
4#zhongfei#19
5#feifei#15
```

2>封装的实体对象

```
@Getter
@Setter
@ToString
public class User {
    private Long id;
    @NotBlank(message = "用户名不能为null或空串")
    private String name;
    private int age;
}
```

3>用于转换大写工具类

```
public class UserServiceImpl {
    public User toUppeCase(User user){
        user.setName(user.getName().toUpperCase());
        return user;
    }
}
```

4>完整代码

```
package com.langfeiyes.batch._29_itemprocessor_composite;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.adapter.ItemProcessorAdapter;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.batch.item.support.CompositeItemProcessor;
import org.springframework.batch.item.validator.BeanValidatingItemProcessor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;

import java.util.Arrays;
import java.util.List;

@SpringBootApplication
@EnableBatchProcessing
public class CompositeProcessorJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;


    @Bean
    public FlatFileItemReader<User> userItemReader(){
        return new FlatFileItemReaderBuilder<User>()
                .name("userItemReader")
                .resource(new ClassPathResource("users-validate.txt"))
                .delimited().delimiter("#")
                .names("id", "name", "age")
                .targetType(User.class)
                .build();
    }

    @Bean
    public ItemWriter<User> itemWriter(){
        return new ItemWriter<User>() {
            @Override
            public void write(List<? extends User> items) throws Exception {
                items.forEach(System.err::println);
            }
        };
    }

    @Bean
    public UserServiceImpl userService(){
        return new UserServiceImpl();
    }
    @Bean
    public BeanValidatingItemProcessor<User> beanValidatingItemProcessor(){
        BeanValidatingItemProcessor<User> beanValidatingItemProcessor = new
BeanValidatingItemProcessor<>();
        beanValidatingItemProcessor.setFilter(true);  //不满足条件丢弃数据
        return beanValidatingItemProcessor;
    }

    @Bean
    public ItemProcessorAdapter<User, User> itemProcessorAdapter(){
        ItemProcessorAdapter<User, User> adapter = new ItemProcessorAdapter<>();
        adapter.setTargetObject(userService());
        adapter.setTargetMethod("toUppeCase");

        return adapter;
    }

    @Bean
    public CompositeItemProcessor<User, User>  compositeItemProcessor(){
        CompositeItemProcessor<User, User> compositeItemProcessor = new
CompositeItemProcessor<>();
        compositeItemProcessor.setDelegates(Arrays.asList(
                beanValidatingItemProcessor(), itemProcessorAdapter()
        ));
        return  compositeItemProcessor;
    }

    @Bean
    public Step step(){
        return stepBuilderFactory.get("step1")
                .<User, User>chunk(1)
                .reader(userItemReader())
                .processor(compositeItemProcessor())
                .writer(itemWriter())
                .build();

    }

    @Bean
    public Job job(){
        return jobBuilderFactory.get("composite-processor-job")
                .start(step())
                .build();
    }
    public static void main(String[] args) {
        SpringApplication.run(CompositeProcessorJob.class, args);
    }
}
```

解析：

核心代码：compositeItemProcessor() 实例方法，使用setDelegates 操作将其他ItemProcessor 处理合并成一个。

10.2 自定义ItemProcessor处理器

除去上面默认的几种处理器外，Spring Batch 也允许我们自定义，具体做法只需要实现ItemProcessor接口即可

需求：自定义处理器，筛选出id为偶数的用户

1>定义读取文件user.txt

```
1#dafei#18
2#xiaofei#16
3#laofei#20
4#zhongfei#19
5#feifei#15
```

2>定义实体对象

```
@Getter
@Setter
@ToString
public class User {
    private Long id;
    private String name;
    private int age;
}
```

3>自定义处理器

```
//自定义
public class CustomizeItemProcessor implements ItemProcessor<User,User> {
    @Override
    public User process(User item) throws Exception {
        //id 为偶数的用户放弃
        //返回null时候 读入的item会被放弃，不会进入itemwriter
        return item.getId() % 2 != 0 ? item : null;
    }
}
```

4>完整代码

```
package com.langfeiyes.batch._30_itemprocessor_customize;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
```

```java
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;

import java.util.List;

@SpringBootApplication
@EnableBatchProcessing
public class CustomizeProcessorJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;


    @Bean
    public FlatFileItemReader<User> userItemReader(){
        return new FlatFileItemReaderBuilder<User>()
                .name("userItemReader")
                .resource(new ClassPathResource("users.txt"))
                .delimited().delimiter("#")
                .names("id", "name", "age")
                .targetType(User.class)
                .build();
    }

    @Bean
    public ItemWriter<User> itemWriter(){
        return new ItemWriter<User>() {
            @Override
            public void write(List<? extends User> items) throws Exception {
                items.forEach(System.err::println);
            }
        };
    }
    @Bean
    public CustomizeItemProcessor customizeItemProcessor(){
        return new CustomizeItemProcessor();
    }

    @Bean
    public Step step(){
        return stepBuilderFactory.get("step1")
                .<User, User>chunk(1)
                .reader(userItemReader())
                .processor(customizeItemProcessor())
                .writer(itemWriter())
                .build();
    }
    @Bean
    public Job job(){
        return jobBuilderFactory.get("customize-processor-job")
                .start(step())
                .build();
    }
    public static void main(String[] args) {
        SpringApplication.run(CustomizeProcessorJob.class, args);
    }
}
```

十一、ItemWriter

有输入那肯定有输出，前面讲了输入ItemReader，接下来就看本篇的输出器：ItemWriter， Spring Batch提供的数据输出组件与数据输入组件是成对，也就是说有啥样子的输入组件，就有啥样子的输出组件。

11.1 输出平面文件

当将读入的数据输出到纯文本文件时，可以通过FlatFileItemWriter 输出器实现。

需求：将user.txt中数据读取出来，输出到outUser.txt文件中

1>定义user.txt文件

```
1#dafei#18
2#xiaofei#16
3#laofei#20
4#zhongfei#19
5#feifei#15
```

2>定义实体对象

```
@Getter
@Setter
@ToString
public class User {
    private Long id;
    private String name;
    private int age;
}
```

3>实现代码

```
package com.langfeiyes.batch._31_itemwriter_flat;


import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
```

```java
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.FlatFileItemWriter;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.batch.item.file.builder.FlatFileItemWriterBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.PathResource;

import java.util.List;

@SpringBootApplication
@EnableBatchProcessing
public class FlatWriteJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Bean
    public FlatFileItemReader<User> userItemReader(){
        return new FlatFileItemReaderBuilder<User>()
                .name("userItemReader")
                .resource(new ClassPathResource("users.txt"))
                .delimited().delimiter("#")
                .names("id", "name", "age")
                .targetType(User.class)
                .build();
    }

    @Bean
    public FlatFileItemWriter<User> itemWriter(){
        return new FlatFileItemWriterBuilder<User>()
                .name("userItemWriter")
                .resource(new PathResource("c:/outUser.txt"))  //输出的文件
                .formatted()  //数据格式指定
                .format("id: %s,姓名：%s,年龄：%s")  //输出数据格式
                .names("id", "name", "age")  //需要输出属性
                .build();
    }

    @Bean
    public Step step(){
        return stepBuilderFactory.get("step1")
                .<User, User>chunk(1)
                .reader(userItemReader())
                .writer(itemWriter())
                .build();

    }

    @Bean
    public Job job(){
        return jobBuilderFactory.get("flat-writer-job")
                .start(step())
                .build();
    }
    public static void main(String[] args) {
        SpringApplication.run(FlatWriteJob.class, args);
    }
}
```

解析：

上面代码核心是itemWriter() 方法，设置到itemWrite读取器配置与输出

```
id: 1,姓名：dafei,年龄：18
id: 2,姓名：xiaofei,年龄：16
id: 3,姓名：laofei,年龄：20
id: 4,姓名：zhongfei,年龄：19
id: 5,姓名：feifei,年龄：15
```

一些拓展

```
@Bean
public FlatFileItemWriter<User> itemWriter(){
    return new FlatFileItemWriterBuilder<User>()
        .name("userItemWriter")
        .resource(new PathResource("c:/outUser.txt"))  //输出的文件
        .formatted()  //数据格式指定
        .format("id: %s,姓名：%s,年龄：%s")  //输出数据格式
        .names("id", "name", "age")  //需要输出属性
        .shouldDeleteIfEmpty(true)   //如果读入数据为空，输出时创建文件直接删除
        .shouldDeleteIfExists(true) //如果输出文件已经存在，则删除
        .append(true)  //如果输出文件已经存在， 不删除，直接追加到现有文件中
        .build();
}
```

11.2 输出Json文件

当将读入的数据输出到Json文件时，可以通过JsonFileItemWriter输出器实现。

需求：将user.txt中数据读取出来，输出到outUser.json文件中

沿用上面的user.txt， user对象将数据输出到outUser.json

```
package com.langfeiyes.batch._32_itemwriter_json;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.FlatFileItemWriter;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.batch.item.file.builder.FlatFileItemWriterBuilder;
import org.springframework.batch.item.json.JacksonJsonObjectMarshaller;
```

```java
import org.springframework.batch.item.json.JsonFileItemWriter;
import org.springframework.batch.item.json.builder.JsonFileItemWriterBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.PathResource;

@SpringBootApplication
@EnableBatchProcessing
public class JsonWriteJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Bean
    public FlatFileItemReader<User> userItemReader(){
        return new FlatFileItemReaderBuilder<User>()
                .name("userItemReader")
                .resource(new ClassPathResource("users.txt"))
                .delimited().delimiter("#")
                .names("id", "name", "age")
                .targetType(User.class)
                .build();
    }


    @Bean
    public JacksonJsonObjectMarshaller<User> objectMarshaller(){
        JacksonJsonObjectMarshaller marshaller = new
JacksonJsonObjectMarshaller();
        return marshaller;
    }

    @Bean
    public JsonFileItemWriter<User> itemWriter(){
        return new JsonFileItemWriterBuilder<User>()
                .name("jsonUserItemWriter")
                .resource(new PathResource("c:/outUser.json"))
                .jsonObjectMarshaller(objectMarshaller())
                .build();
    }


    @Bean
    public Step step(){
        return stepBuilderFactory.get("step1")
                .<User, User>chunk(1)
                .reader(userItemReader())
                .writer(itemWriter())
                .build();

    }

    @Bean
    public Job job(){
        return jobBuilderFactory.get("json-writer-job")
                .start(step())
                .build();
    }
    public static void main(String[] args) {
        SpringApplication.run(JsonWriteJob.class, args);
    }
}
```

结果:

```
[
 {"id":1,"name":"dafei","age":18},
 {"id":2,"name":"xiaofei","age":16},
 {"id":3,"name":"laofei","age":20},
 {"id":4,"name":"zhongfei","age":19},
 {"id":5,"name":"feifei","age":15}
]
```

解析:

1>itemWriter() 实例方法构建JsonFileItemWriter 实例，需要明确指定Json格式装配器

2>Spring Batch默认提供装配器有2个：JacksonJsonObjectMarshaller GsonJsonObjectMarshaller 分别对应Jackson 跟 Gson 2种json格式解析逻辑，本案例用的是Jackson

11.3 输出数据库

当将读入的数据需要输出到数据库时，可以通过JdbcBatchItemWriter输出器实现。

需求：将user.txt中数据读取出来，输出到数据库user表中

沿用上面的user.txt， user对象将数据输出到user表中

1>定义操作数据库预编译类

```
//写入数据库需要操作insert sql， 使用预编译就需要明确指定参数值
public class UserPreStatementSetter implements ItemPreparedStatementSetter<User>
{
    @Override
    public void setValues(User item, PreparedStatement ps) throws SQLException {
        ps.setLong(1, item.getId());
        ps.setString(2, item.getName());
        ps.setInt(3, item.getAge());
    }
}
```

2>完整代码

```
package com.langfeiyes.batch._33_itemwriter_db;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
```

```java
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.item.database.JdbcBatchItemWriter;
import
org.springframework.batch.item.database.builder.JdbcBatchItemWriterBuilder;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.batch.item.json.JacksonJsonObjectMarshaller;
import org.springframework.batch.item.json.JsonFileItemWriter;
import org.springframework.batch.item.json.builder.JsonFileItemWriterBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.PathResource;

import javax.sql.DataSource;

@SpringBootApplication
@EnableBatchProcessing
public class JdbcWriteJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Autowired
    private DataSource dataSource;

    @Bean
    public FlatFileItemReader<User> userItemReader(){
        return new FlatFileItemReaderBuilder<User>()
                .name("userItemReader")
                .resource(new ClassPathResource("users.txt"))
                .delimited().delimiter("#")
                .names("id", "name", "age")
                .targetType(User.class)
                .build();
    }
    @Bean
    public UserPreStatementSetter preStatementSetter(){
        return new UserPreStatementSetter();
    }
    @Bean
    public JdbcBatchItemWriter<User>  itemWriter(){
        return new JdbcBatchItemWriterBuilder<User>()
                .dataSource(dataSource)
                .sql("insert into user(id, name, age) values(?,?,?)")
                .itemPreparedStatementSetter(preStatementSetter())
                .build();
    }
    @Bean
    public Step step(){
        return stepBuilderFactory.get("step1")
                .<User, User>chunk(1)
                .reader(userItemReader())
                .writer(itemWriter())
                .build();
    }

    @Bean
    public Job job(){
        return jobBuilderFactory.get("jdbc-writer-job")
                .start(step())
                .build();
    }
    public static void main(String[] args) {
        SpringApplication.run(JdbcWriteJob.class, args);
    }
}
```

解析：

核心代码在itemWriter() 实例方法中， 需要1>准备构建JdbcBatchItemWriter实例 2>提前准备数据，3>准备sql语句 4>准备参数绑定器

11.4 输出多终端

上面几种输出方法都是一对一，真实开发可能没那么简单了，可能存在一对多，多个终端输出，此时怎么办？答案是使用Spring Batch 提供的CompositeItemWriter 组合输出器。

需求：将user.txt中数据读取出来，输出到outUser.txt/outUser.json/数据库user表中

沿用上面的user.txt， user对象将数据输出到outUser.txt/outUser.json/user表中

```
package com.langfeiyes.batch._34_itemwriter_composite;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.item.database.JdbcBatchItemWriter;
import
org.springframework.batch.item.database.builder.JdbcBatchItemWriterBuilder;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.FlatFileItemWriter;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.batch.item.file.builder.FlatFileItemWriterBuilder;
import org.springframework.batch.item.json.JacksonJsonObjectMarshaller;
import org.springframework.batch.item.json.JsonFileItemWriter;
import org.springframework.batch.item.json.builder.JsonFileItemWriterBuilder;
import org.springframework.batch.item.support.CompositeItemWriter;
import
org.springframework.batch.item.support.builder.CompositeItemWriterBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.PathResource;
```

```java
import javax.sql.DataSource;
import java.util.Arrays;

@SpringBootApplication
@EnableBatchProcessing
public class CompositeWriteJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Autowired
    public DataSource dataSource;

    @Bean
    public FlatFileItemReader<User> userItemReader(){
        return new FlatFileItemReaderBuilder<User>()
                .name("userItemReader")
                .resource(new ClassPathResource("users.txt"))
                .delimited().delimiter("#")
                .names("id", "name", "age")
                .targetType(User.class)
                .build();
    }

    @Bean
    public FlatFileItemWriter<User> flatFileItemWriter(){
        return new FlatFileItemWriterBuilder<User>()
                .name("userItemWriter")
                .resource(new PathResource("c:/outUser.txt"))
                .formatted()  //数据格式指定
                .format("id: %s,姓名: %s,年龄: %s")  //输出数据格式
                .names("id", "name", "age")  //需要输出属性
                .build();
    }

    @Bean
    public JacksonJsonObjectMarshaller<User> objectMarshaller(){
        JacksonJsonObjectMarshaller marshaller = new
JacksonJsonObjectMarshaller();
        return marshaller;
    }

    @Bean
    public JsonFileItemWriter<User> jsonFileItemWriter(){
        return new JsonFileItemWriterBuilder<User>()
                .name("jsonUserItemWriter")
                .resource(new PathResource("c:/outUser.json"))
                .jsonObjectMarshaller(objectMarshaller())
                .build();
    }

    @Bean
    public UserPreStatementSetter preStatementSetter(){
        return new UserPreStatementSetter();
    }
    @Bean
    public JdbcBatchItemWriter<User> jdbcBatchItemWriter(){
        return new JdbcBatchItemWriterBuilder<User>()
                .dataSource(dataSource)
                .sql("insert into user(id, name, age) values(?,?,?)")
                .itemPreparedStatementSetter(preStatementSetter())
                .build();
    }

    @Bean
    public CompositeItemWriter<User> compositeItemWriter(){
        return new CompositeItemWriterBuilder<User>()
                .delegates(Arrays.asList(flatFileItemWriter(),
jsonFileItemWriter(), jdbcBatchItemWriter()))
                .build();
    }


    @Bean
    public Step step(){
        return stepBuilderFactory.get("step1")
                .<User, User>chunk(1)
                .reader(userItemReader())
                .writer(compositeItemWriter())
                .build();

    }

    @Bean
    public Job job(){
        return jobBuilderFactory.get("composite-writer-job")
                .start(step())
                .build();
    }
    public static void main(String[] args) {
        SpringApplication.run(CompositeWriteJob.class, args);
    }
}
```

解析：

代码没有啥技术难度，都是将前面的几种方式通过CompositeItemWriter 类整合在一起

```
@Bean
public CompositeItemWriter<User> compositeItemWriter(){
    return new CompositeItemWriterBuilder<User>()
        .delegates(Arrays.asList(flatFileItemWriter(), jsonFileItemWriter(),
jdbcBatchItemWriter()))
        .build();
}
```

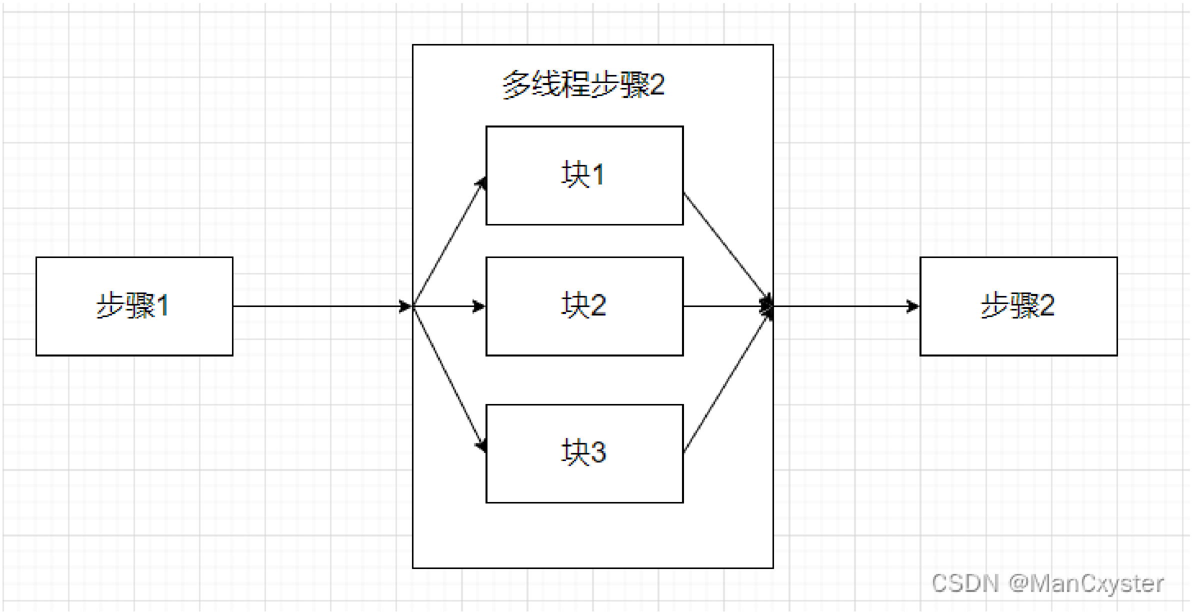
十二、Spring Batch 高级

前面讲的Spring Batch 基本上能满足日常批处理了，下面则是Spring Batch 高级部分内容，大家可以自己选择了解。

12.1 多线程步骤

默认的情况下，步骤基本上在单线程中执行，那能不能在多线程环境执行呢？答案肯定是yes，但是也要注意，多线程环境步骤执行一定要慎重。原因：**多线程环境下，步骤是要设置不可重启。**

Spring Batch 的多线程步骤是使用Spring 的 TaskExecutor(任务执行器)实现的。约定每一个块开启一个线程独立执行。



需求：分5个块处理user-thread.txt文件

1>编写user-thread.txt文件

```
1#dafei#18
2#xiaofei#16
3#laofei#20
4#zhongfei#19
5#feifei#15
6#zhangsan#14
7#lisi#13
8#wangwu#12
9#zhaoliu#11
10#qianqi#10
```

2>定义实体对象

```
@Getter
@Setter
@ToString
public class User {
    private Long id;
    private String name;
    private int age;
}
```

3>完整代码

```
package com.langfeiyes.batch._35_step_thread;
```

```java
import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.task.SimpleAsyncTaskExecutor;

import java.util.List;

@SpringBootApplication
@EnableBatchProcessing
public class ThreadStepJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Bean
    public FlatFileItemReader<User> userItemReader(){

        System.out.println(Thread.currentThread());

        FlatFileItemReader<User> reader = new FlatFileItemReaderBuilder<User>()
                .name("userItemReader")
                .saveState(false) //防止状态被覆盖
                .resource(new ClassPathResource("user-thread.txt"))
                .delimited().delimiter("#")
                .names("id", "name", "age")
                .targetType(User.class)
                .build();

        return reader;
    }

    @Bean
    public ItemWriter<User> itemWriter(){
        return new ItemWriter<User>() {
            @Override
            public void write(List<? extends User> items) throws Exception {
                items.forEach(System.err::println);
            }
        };
    }

    @Bean
    public Step step(){
        return stepBuilderFactory.get("step1")
```

```
                .<User, User>chunk(2)
                .reader(userItemReader())
                .writer(itemWriter())
                .taskExecutor(new SimpleAsyncTaskExecutor())
                .build();

    }

    @Bean
    public Job job(){
        return jobBuilderFactory.get("thread-step-job")
                .start(step())
                .build();
    }
    public static void main(String[] args) {
        SpringApplication.run(ThreadStepJob.class, args);
    }
}
```

4>结果

```
User(id=2, name=xiaofei, age=16)
User(id=5, name=feifei, age=15)
User(id=4, name=zhongfei, age=19)
User(id=7, name=lisi, age=13)
User(id=1, name=dafei, age=18)
User(id=6, name=zhangsan, age=14)
User(id=3, name=laofei, age=20)
User(id=8, name=wangwu, age=12)
User(id=9, name=zhaoliu, age=11)
User(id=10, name=qianqi, age=10)
```

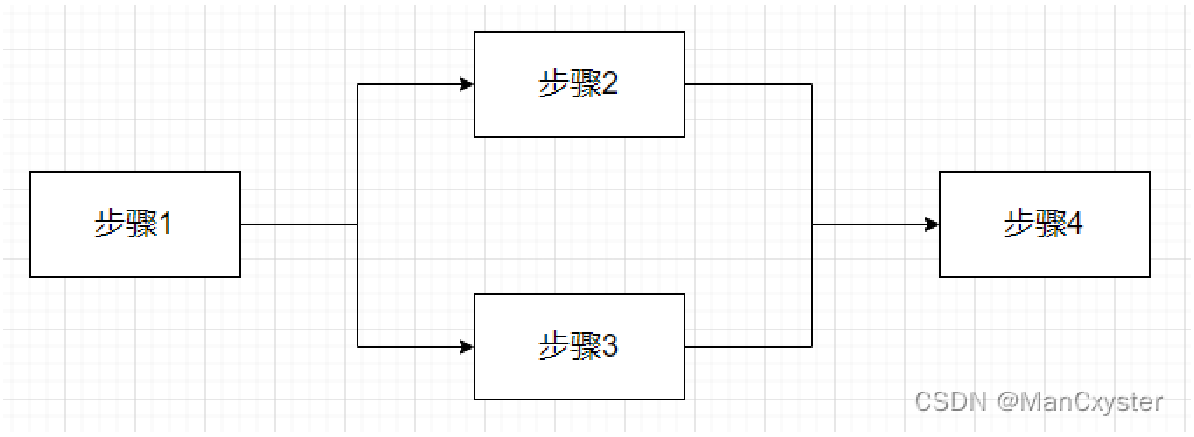
解析

1：**userItemReader()** 加上**saveState(false)** Spring Batch 提供大部分的ItemReader是有状态的，作业重启基本通过状态来确定作业停止位置，而在多线程环境中，如果对象维护状态被多个线程访问，可能存在线程间状态相互覆盖问题。所以设置为false表示关闭状态，但这也意味着作业不能重启了。

2：**step()** 方法加上**.taskExecutor(new SimpleAsyncTaskExecutor())** 为作业步骤添加了多线程处理能力，以块为单位，一个块一个线程，观察上面的结果，很明显能看出输出的顺序是乱序的。改变 job 的名字再执行，会发现输出数据每次都不一样。

12.2 并行步骤

并行步骤，指的是某2个或者多个步骤同时执行。比如下图



图中，流程从步骤1执行，然后执行步骤2， 步骤3，当步骤2/3执行结束之后，在执行步骤4.

设想一种场景，当读取2个或者多个互不关联的文件时，可以多个文件同时读取，这个就是并行步骤。

需求：现有user-parallel.txt, user-parallel.json 2个文件将它们中数据读入内存

1>编写user-parallel.txt, user-parallel.json

```
6#zhangsan#14
7#lisi#13
8#wangwu#12
9#zhaoliu#11
10#qianqi#10
```

```
[
  {"id":1, "name":"dafei", "age":18},
  {"id":2, "name":"xiaofei", "age":17},
  {"id":3, "name":"zhongfei", "age":16},
  {"id":4, "name":"laofei", "age":15},
  {"id":5, "name":"feifei", "age":14}
]
```

2>编写实体对象

```
@Getter
@Setter
@ToString
public class User {
    private Long id;
    private String name;
    private int age;
}
```

3>代码实现

```
package com.langfeiyes.batch._36_step_parallel;


import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
```

```java
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.job.builder.FlowBuilder;
import org.springframework.batch.core.job.flow.Flow;
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.batch.item.json.JacksonJsonObjectReader;
import org.springframework.batch.item.json.JsonItemReader;
import org.springframework.batch.item.json.builder.JsonItemReaderBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.task.SimpleAsyncTaskExecutor;

import java.util.List;

@SpringBootApplication
@EnableBatchProcessing
public class ParallelStepJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;


    @Bean
    public JsonItemReader<User> jsonItemReader(){
        ObjectMapper objectMapper = new ObjectMapper();
        JacksonJsonObjectReader<User> jsonObjectReader = new
JacksonJsonObjectReader<>(User.class);
        jsonObjectReader.setMapper(objectMapper);

        return new JsonItemReaderBuilder<User>()
                .name("userJsonItemReader")
                .jsonObjectReader(jsonObjectReader)
                .resource(new ClassPathResource("user-parallel.json"))
                .build();
    }

    @Bean
    public FlatFileItemReader<User> flatItemReader(){
        return new FlatFileItemReaderBuilder<User>()
                .name("userItemReader")
                .resource(new ClassPathResource("user-parallel.txt"))
                .delimited().delimiter("#")
                .names("id", "name", "age")
                .targetType(User.class)
                .build();
    }
    @Bean
    public ItemWriter<User> itemWriter(){
        return new ItemWriter<User>() {
            @Override
            public void write(List<? extends User> items) throws Exception {
                items.forEach(System.err::println);
            }
        };
    }

    @Bean
    public Step jsonStep(){
        return stepBuilderFactory.get("jsonStep")
                .<User, User>chunk(2)
                .reader(jsonItemReader())
                .writer(itemWriter())
                .build();
    }

    @Bean
    public Step flatStep(){
        return stepBuilderFactory.get("step2")
                .<User, User>chunk(2)
                .reader(flatItemReader())
                .writer(itemWriter())
                .build();
    }

    @Bean
    public Job parallelJob(){

        //线程1-读user-parallel.txt
        Flow parallelFlow1 = new FlowBuilder<Flow>("parallelFlow1")
                .start(flatStep())
                .build();

        //线程2-读user-parallel.json
        Flow parallelFlow2 = new FlowBuilder<Flow>("parallelFlow2")
                .start(jsonStep())
                .split(new SimpleAsyncTaskExecutor())
                .add(parallelFlow1)
                .build();


        return jobBuilderFactory.get("parallel-step-job")
                .start(parallelFlow2)
                .end()
                .build();
    }
    public static void main(String[] args) {
        SpringApplication.run(ParallelStepJob.class, args);
    }
}
```

结果

```
User(id=6, name=zhangsan, age=14)
User(id=7, name=lisi, age=13)
User(id=8, name=wangwu, age=12)
User(id=9, name=zhaoliu, age=11)
User(id=1, name=dafei, age=18)
User(id=2, name=xiaofei, age=17)
User(id=10, name=qianqi, age=10)
User(id=3, name=zhongfei, age=16)
User(id=4, name=laofei, age=15)
User(id=5, name=feifei, age=14)
```

解析：

1：jsonItemReader() flatItemReader() 定义2个读入操作，分别读json格式跟普通文本格式

2：parallelJob() 配置job，需要指定并行的flow步骤，先是parallelFlow1然后是parallelFlow2， 2个步骤间使用**.split(new SimpleAsyncTaskExecutor())** 隔开，表示线程池开启2个线程，分别处理 parallelFlow1， parallelFlow2 2个步骤。

12.3 分区步骤

分区：有划分，区分意思，在SpringBatch 分区步骤讲的是给执行步骤区分上下级。

上级： 主步骤(Master Step)

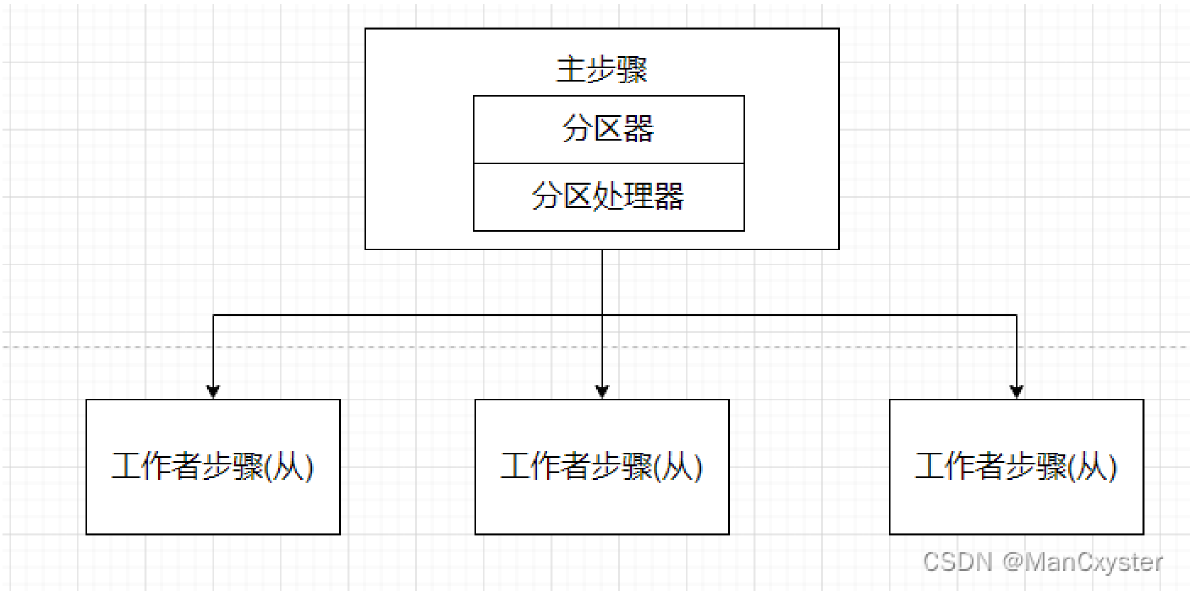
下级： 从步骤--工作步骤(Work Step)

主步骤是领导，不用干活，负责管理从步骤，从步骤是下属，必须干活。

一个主步骤下辖管理多个从步骤。

注意： 从步骤，不管多小，它也一个完整的Spring Batch 步骤，负责各自的读入、处理、写入等。

分区步骤结构图



分区步骤一般用于海量数据的处理上，其采用是分治思想。主步骤将大的数据划分多个小的数据集，然后开启多个从步骤，要求每个从步骤负责一个数据集。当所有从步骤处理结束，整作业流程才算结束。

分区器

主步骤核心组件，负责数据分区，将完整的数据拆解成多个数据集，然后指派给从步骤，让其执行。

拆分规则由Partitioner分区器接口定制，默认的实现类：**MultiResourcePartitioner**

```java
public interface Partitioner {
    Map<String, ExecutionContext> partition(int gridSize);
}
```

Partitioner 接口只有唯一的方法：partition 参数gridSize表示要分区的大小，可以理解为要开启多个worker步骤，返回值是一个Map， 其中key：worker步骤名称， value：worker步骤启动需要参数值，一般包含分区元数据，比如起始位置，数据量等。

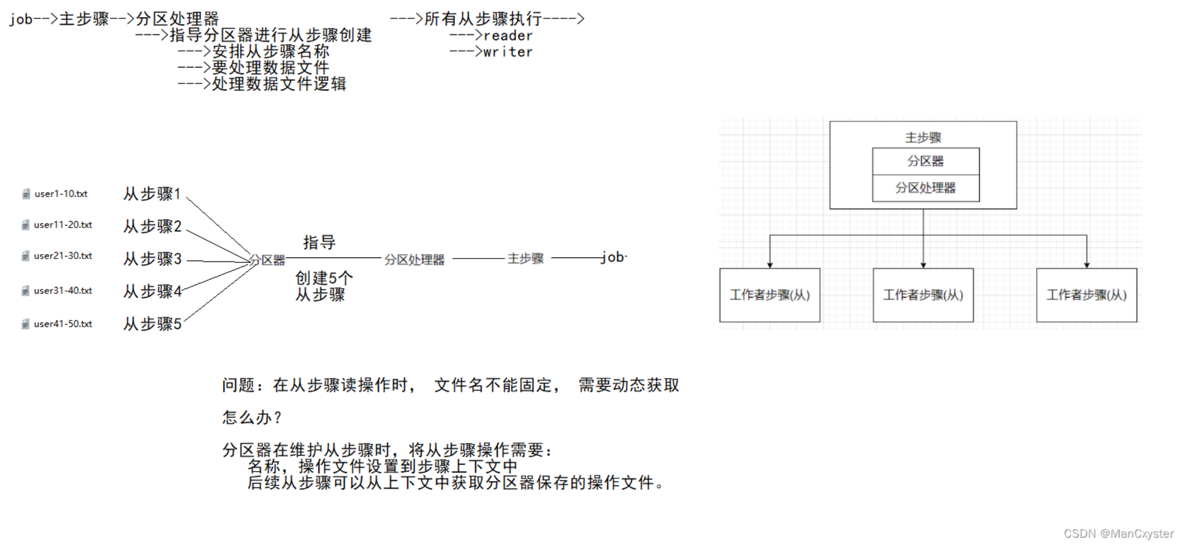
分区处理器

主步骤核心组件，统一管理work 步骤， 并给work步骤指派任务。

管理规则由PartitionHandler 接口定义，默认的实现类：**TaskExecutorPartitionHandler**

需求：下面几个文件将数据读入内存

分析：



```
job-->主步骤-->分区处理器                    --->所有从步骤执行---->
          --->指导分区器进行从步骤创建          --->reader
              --->安排从步骤名称                --->writer
              --->要处理数据文件
              --->处理数据文件逻辑
```

问题：在从步骤读操作时， 文件名不能固定， 需要动态获取

怎么办？

分区器在维护从步骤时，将从步骤操作需要：
  名称，操作文件设置到步骤上下文中
  后续从步骤可以从上下文中获取分区器保存的操作文件。

步骤1：准备数据

user1-10.txt

```
1#dafei#18
2#dafei#18
3#dafei#18
4#dafei#18
5#dafei#18
6#dafei#18
7#dafei#18
8#dafei#18
9#dafei#18
10#dafei#18
```

user11-20.txt

```
11#dafei#18
12#dafei#18
13#dafei#18
14#dafei#18
15#dafei#18
16#dafei#18
17#dafei#18
18#dafei#18
19#dafei#18
20#dafei#18
```

user21-30.txt

```
21#dafei#18
22#dafei#18
23#dafei#18
24#dafei#18
25#dafei#18
26#dafei#18
27#dafei#18
28#dafei#18
29#dafei#18
30#dafei#18
```

user31-40.txt

```
31#dafei#18
32#dafei#18
33#dafei#18
34#dafei#18
35#dafei#18
36#dafei#18
37#dafei#18
38#dafei#18
39#dafei#18
40#dafei#18
```

user41-50.txt

```
41#dafei#18
42#dafei#18
43#dafei#18
44#dafei#18
45#dafei#18
46#dafei#18
47#dafei#18
48#dafei#18
49#dafei#18
50#dafei#18
```

步骤2：准备实体类

```
@Getter
@Setter
@ToString
public class User {
    private Long id;
    private String name;
    private int age;
}
```

步骤3：配置分区逻辑

```
public class UserPartitioner  implements Partitioner {
    @Override
    public Map<String, ExecutionContext> partition(int gridSize) {
        Map<String, ExecutionContext> result = new HashMap<>(gridSize);

        int range = 10; //文件间隔
        int start = 1; //开始位置
        int end = 10;  //结束位置
        String text = "user%s-%s.txt";

        for (int i = 0; i < gridSize; i++) {
            ExecutionContext value = new ExecutionContext();
            Resource resource = new ClassPathResource(String.format(text, start,
end));
            try {
                value.putString("file", resource.getURL().toExternalForm());
            } catch (IOException e) {
                e.printStackTrace();
            }
            start += range;
            end += range;

            result.put("user_partition_" + i, value);
        }
        return result;
    }
}
```

步骤4：全部代码

```
package com.langfeiyes.batch._37_step_part;

import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.EnableBatchProcessing;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.configuration.annotation.StepScope;
import org.springframework.batch.core.partition.PartitionHandler;
import
org.springframework.batch.core.partition.support.MultiResourcePartitioner;
```

```java
import
org.springframework.batch.core.partition.support.TaskExecutorPartitionHandler;
import org.springframework.batch.item.ExecutionContext;
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.io.ClassPathResource;
import org.springframework.core.io.Resource;
import org.springframework.core.task.SimpleAsyncTaskExecutor;

import java.util.List;

@SpringBootApplication
@EnableBatchProcessing
public class PartStepJob {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;



    //每个分区文件读取
    @Bean
    @StepScope
    public FlatFileItemReader<User> flatItemReader(@Value("#
{stepExecutionContext['file']}") Resource resource){
        return new FlatFileItemReaderBuilder<User>()
                .name("userItemReader")
                .resource(resource)
                .delimited().delimiter("#")
                .names("id", "name", "age")
                .targetType(User.class)
                .build();
    }

    @Bean
    public ItemWriter<User> itemWriter(){
        return new ItemWriter<User>() {
            @Override
            public void write(List<? extends User> items) throws Exception {
                items.forEach(System.err::println);
            }
        };
    }


    //文件分区器-设置分区规则
    @Bean
    public UserPartitioner userPartitioner(){
        return new UserPartitioner();
    }
    //文件分区处理器-处理分区
    @Bean
    public PartitionHandler userPartitionHandler() {
        TaskExecutorPartitionHandler handler = new
TaskExecutorPartitionHandler();
        handler.setGridSize(5);
        handler.setTaskExecutor(new SimpleAsyncTaskExecutor());
        handler.setStep(workStep());
        try {
            handler.afterPropertiesSet();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return handler;
    }

    //每个从分区操作步骤
    @Bean
    public Step workStep() {
        return stepBuilderFactory.get("workStep")
                .<User, User>chunk(10)
                .reader(flatItemReader(null))
                .writer(itemWriter())
                .build();
    }

    //主分区操作步骤
    @Bean
    public Step masterStep() {
        return stepBuilderFactory.get("masterStep")
                .partitioner(workStep().getName(),userPartitioner())
                .partitionHandler(userPartitionHandler())
                .build();
    }


    @Bean
    public Job partJob(){
        return jobBuilderFactory.get("part-step-job")
                .start(masterStep())
                .build();
    }
    public static void main(String[] args) {
        SpringApplication.run(PartStepJob.class, args);
    }
}
```

结果:

```
User(id=31, name=dafei, age=18)
User(id=32, name=dafei, age=18)
User(id=33, name=dafei, age=18)
User(id=34, name=dafei, age=18)
User(id=35, name=dafei, age=18)
User(id=36, name=dafei, age=18)
User(id=37, name=dafei, age=18)
User(id=38, name=dafei, age=18)
User(id=39, name=dafei, age=18)
User(id=40, name=dafei, age=18)
User(id=41, name=dafei, age=18)
User(id=42, name=dafei, age=18)
User(id=43, name=dafei, age=18)
User(id=44, name=dafei, age=18)
User(id=45, name=dafei, age=18)
User(id=46, name=dafei, age=18)
User(id=47, name=dafei, age=18)
User(id=48, name=dafei, age=18)
User(id=49, name=dafei, age=18)
User(id=50, name=dafei, age=18)
User(id=21, name=dafei, age=18)
User(id=22, name=dafei, age=18)
User(id=23, name=dafei, age=18)
User(id=24, name=dafei, age=18)
User(id=25, name=dafei, age=18)
User(id=26, name=dafei, age=18)
User(id=27, name=dafei, age=18)
User(id=28, name=dafei, age=18)
User(id=29, name=dafei, age=18)
User(id=30, name=dafei, age=18)
User(id=1, name=dafei, age=18)
User(id=2, name=dafei, age=18)
User(id=3, name=dafei, age=18)
User(id=4, name=dafei, age=18)
User(id=5, name=dafei, age=18)
User(id=6, name=dafei, age=18)
User(id=7, name=dafei, age=18)
User(id=8, name=dafei, age=18)
User(id=9, name=dafei, age=18)
User(id=10, name=dafei, age=18)
User(id=11, name=dafei, age=18)
User(id=12, name=dafei, age=18)
User(id=13, name=dafei, age=18)
User(id=14, name=dafei, age=18)
User(id=15, name=dafei, age=18)
User(id=16, name=dafei, age=18)
User(id=17, name=dafei, age=18)
User(id=18, name=dafei, age=18)
User(id=19, name=dafei, age=18)
User(id=20, name=dafei, age=18)
```

解析：核心点

1>文件分区器：userPartitioner()， 分别加载5个文件进入到程序

2>文件分区处理器：userPartitionHandler() ，指定要分几个区，由谁来处理

3>分区从步骤：workStep() 指定读逻辑与写逻辑

4>分区文件读取：flatItemReader()，需要传入Resource对象，这个对象在userPartitioner()已经标记为file

5>分区主步骤：masterStep() ，指定分区名称与分区器，指定分区处理器

十三、综合案例

到这，整个Spring Batch 教程知识点就全部讲完了，接下来就使用一个综合案例将讲过核心知识串联起来，再来回顾一遍。

13.1 案例需求

1>先动态生成50w条员工数据，存放在employee.csv文件中

2>启动作业异步读取employee.csv文件，将读到数据写入到employee_temp表，要求记录读与写消耗时间

3>使用分区的方式将employee_temp表的数据读取并写入到employee表

13.2 分析

上面需求存在一定连贯性，为了操作简单，使用springMVC项目， 每一个需求对应一个接口：

1：发起 **/dataInit** 初始化50w数据进入employee.csv文件

使用技术点：SpringMVC IO

2：发起**/csvToDB** 启动作业，将employee.csv 数据写入employee_temp表, 记录读与写消耗时间

使用技术点：SpringMVC ItemReader JobExecutionListener

ItemWriter (如果使用Mybatis框架MyBatisBatchItemWriter/MyBatisPagingItemReaderReader)

3：发起**/dbToDB** 启动作业，将employee_temp数据写入employee表

使用技术点：SpringMVC ItemReader partitioner

ItemWriter(如果使用Mybatis框架：MyBatisBatchItemWriter/MyBatisPagingItemReaderReader)

13.3 项目准备

步骤1：新开spring-batch-example

步骤2：导入依赖

```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.3</version>
    <relativePath/>
</parent>
<properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-batch</artifactId>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.12</version>
    </dependency>


    <dependency>
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
        <version>1.3.2</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid-spring-boot-starter</artifactId>
        <version>1.1.14</version>
    </dependency>

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
</dependencies>
```

步骤3：配置文件

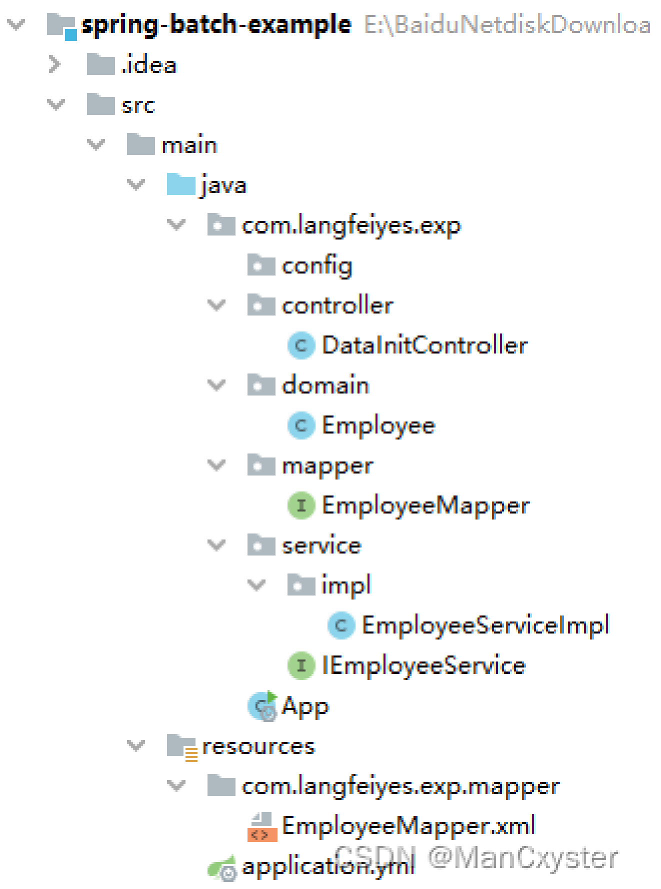
```yaml
spring:
  datasource:
    username: root
    password: admin
    url: jdbc:mysql://127.0.0.1:3306/springbatch?
serverTimezone=GMT%2B8&useSSL=false&allowPublicKeyRetrieval=true
    driver-class-name: com.mysql.cj.jdbc.Driver
    # 初始化数据库，文件在依赖jar包中
  sql:
    init:
      schema-locations: classpath:org/springframework/batch/core/schema-
mysql.sql
      #mode: always
      mode: never
  batch:
    job:
      enabled: false

  druid:
    # 连接池配置
    #初始化连接池的连接数量 大小，最小，最大
    initial-size: 10
    min-idle: 10
    max-active: 20
    #配置获取连接等待超时的时间
    max-wait: 60000
    #配置间隔多久才进行一次检测，检测需要关闭的空闲连接，单位是毫秒
    time-between-eviction-runs-millis: 60000
    # 配置一个连接在池中最小生存的时间，单位是毫秒
    min-evictable-idle-time-millis: 30000
    validation-query: SELECT 1 FROM DUAL
    test-while-idle: true
    test-on-borrow: true
    test-on-return: false
    # 是否缓存preparedStatement，也就是PSCache  官方建议MySQL下建议关闭   个人建议如果想
用SQL防火墙 建议打开
    pool-prepared-statements: false
    max-pool-prepared-statement-per-connection-size: 20

mybatis:
  configuration:
    default-executor-type: batch


job:
  data:
    path: D:/spring-batch-example/
```

步骤4：建立employee表与employe_temp表

```
CREATE TABLE `employee` (
  `id` int NOT NULL AUTO_INCREMENT,
  `name` varchar(255) DEFAULT NULL,
  `age` int DEFAULT NULL,
  `sex` int DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3;
```

```
CREATE TABLE `employee_temp` (
  `id` int NOT NULL AUTO_INCREMENT,
  `name` varchar(255) DEFAULT NULL,
  `age` int DEFAULT NULL,
  `sex` int DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb3;
```

步骤5：建立基本代码体系-domain-mapper-service-controller-mapper.xml



```
spring-batch-example E:\BaiduNetdiskDownloa
  .idea
  src
    main
      java
        com.langfeiyes.exp
          config
          controller
            DataInitController
          domain
            Employee
          mapper
            EmployeeMapper
          service
            impl
              EmployeeServiceImpl
            IEmployeeService
          App
      resources
        com.langfeiyes.exp.mapper
          EmployeeMapper.xml
        application.yml
```

domain

```java
@Setter
@Getter
@ToString
public class Employee {
    private Long id;
    private String name;
    private int age;
    private int sex;
}
```

mapper.java

```java
public interface EmployeeMapper {

    /**
     * 添加
     */
    int save(Employee employee);
}
```

service接口

```
public interface IEmployeeService {
    /**
     * 保存
     */
    void save(Employee employee);
}
```

service接口实现类

```
@Service
public class EmployeeServiceImpl implements IEmployeeService {
    @Autowired
    private EmployeeMapper employeeMapper;
    @Override
    public void save(Employee employee) {
        employeeMapper.save(employee);
    }
}
```

启动类

```
@SpringBootApplication
@EnableBatchProcessing
@MapperScan("com.langfeiyes.exp.mapper")
public class App {

    public static void main(String[] args) {
        SpringApplication.run(App.class, args);
    }
}
```

Mapper.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.langfeiyes.exp.mapper.EmployeeMapper">

    <resultMap id="BaseResultMap" type="com.langfeiyes.exp.domain.Employee">
        <result column="id" jdbcType="INTEGER" property="id" />
        <result column="name" jdbcType="VARCHAR" property="name" />
        <result column="age" jdbcType="VARCHAR" property="age" />
        <result column="sex" jdbcType="VARCHAR" property="sex" />
    </resultMap>


    <insert id="save" keyColumn="id" useGeneratedKeys="true" keyProperty="id">
        insert into employee(id, name, age, sex) values(#{id},#{name},#{age},#
{sex})
    </insert>
</mapper>
```

13.4 需求一

需求：先动态生成50w条员工数据，存放再employee.csv文件中

步骤1：定义：DataInitController

```
@RestController
public class DataInitController {

    @Autowired
    private IEmployeeService employeeService;

    @GetMapping("/dataInit")
    public String dataInit() throws IOException {
        employeeService.dataInit();
        return "ok";
    }
}
```

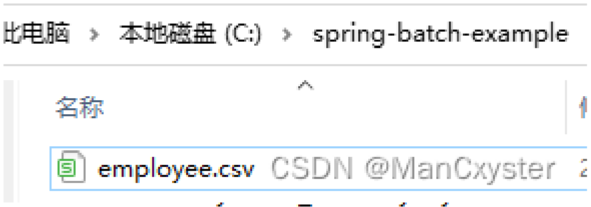
步骤2：在IEmployeeService 添加dataInit 方法

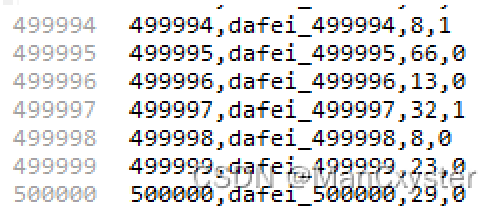
```
public interface IEmployeeService {
    /**
     * 保存
     */
    void save(Employee employee);

    /**
     * 初始化数据：生成50w数据
     */
    void dataInit() throws IOException;
}
```

步骤3：在EmployeeServiceImpl 实现方法

```
@Service
public class EmployeeServiceImpl implements IEmployeeService {
    @Autowired
    private EmployeeMapper employeeMapper;
    @Override
    public void save(Employee employee) {
        employeeMapper.save(employee);
    }

    @Value("${job.data.path}")
    public String path;

    @Override
    public void dataInit() throws IOException {
        File file = new File(path, "employee.csv");
        if (file.exists()) {
            file.delete();
        }
        file.createNewFile();
        FileOutputStream out = new FileOutputStream(file);
        String txt = "";
        Random ageR = new Random();
        Random boolR = new Random();

        // 给文件中生产50万条数据
        long beginTime = System.currentTimeMillis();
        System.out.println("开始时间：【 " + beginTime + " 】");
        for (int i = 1; i <= 500000; i++) {
            if(i == 500000){
                txt = i+",dafei_"+ i +"," + ageR.nextInt(100) + "," +
(boolR.nextBoolean()?1:0);
            }else{
                txt = i+",dafei_"+ i +"," + ageR.nextInt(100) + "," +
(boolR.nextBoolean()?1:0) +"\n";
            }

            out.write(txt.getBytes());
            out.flush();
        }
        out.close();
        System.out.println("总共耗时：【 " + (System.currentTimeMillis() -
beginTime) + " 】毫秒");
    }
}
```

步骤4：访问http://localhost:8080/dataInit 生成数据。





```
499994  499994,dafei_499994,8,1
499995  499995,dafei_499995,66,0
499996  499996,dafei_499996,13,0
499997  499997,dafei_499997,32,1
499998  499998,dafei_499998,8,0
499999  499999,dafei_499999,23,0
500000  500000,dafei_500000,29,0
```

13.5 需求二

需求：启动作业异步读取employee.csv文件，将读到数据写入到employee_temp表，要求记录读与写消耗时间

步骤1：修改IEmployeeService 接口

```
public interface IEmployeeService {
    /**
     * 保存
     */
    void save(Employee employee);

    /**
     * 初始化数据：生成50w数据
     */
    void dataInit() throws IOException;
    /**
     * 清空数据
     */
    void truncateAll();

    /**
     * 清空employee_temp数据
     */
    void truncateTemp();
}
```

步骤2：修改EmployeeServiceImpl

```
@Override
public void truncateAll() {
    employeeMapper.truncateAll();
}

@Override
public void truncateTemp() {
    employeeMapper.truncateTemp();
}
```

步骤3：修改IEmployeeMapper.java

```
public interface EmployeeMapper {

    /**
     * 添加
     */
    int save(Employee employee);

    /**
     * 添加临时表
     * @param employee
     * @return
     */
    int saveTemp(Employee employee);

    /**
     * 清空数据
     */
    void truncateAll();

    /**
     * 清空临时表数据
     */
    void truncateTemp();
}
```

步骤4：修改EmployeeMapper.xml

```
<insert id="saveTemp" keyColumn="id" useGeneratedKeys="true" keyProperty="id">
    insert into employee_temp(id, name, age, sex) values(#{id},#{name},#{age},#
{sex})
</insert>

<delete id="truncateAll">
    truncate employee
</delete>

<delete id="truncateTemp">
    truncate employee_temp
</delete>
```

步骤5：在com.langfeiyes.exp.job.listener 包新建监听器，用于计算开始结束时间

```
package com.langfeiyes.exp.job.listener;

import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.JobExecutionListener;


public class CsvToDBJobListener implements JobExecutionListener {

    @Override
    public void beforeJob(JobExecution jobExecution) {
            long begin = System.currentTimeMillis();
            jobExecution.getExecutionContext().putLong("begin", begin);
            System.err.println("-------------------------【CsvToDBJob开始时间：】--
-->"+begin+"<-----------------------------");
        }

    @Override
    public void afterJob(JobExecution jobExecution) {
                long begin =
jobExecution.getExecutionContext().getLong("begin");
                long end = System.currentTimeMillis();
                System.err.println("-------------------------【CsvToDBJob结束时
间：】---->"+end+"<-----------------------------");
                System.err.println("-------------------------【CsvToDBJob总耗
时：】---->"+(end - begin)+"<-----------------------------");
            }
}
```

步骤6：在com.langfeiyes.exp.job.config包定义CsvToDBJobConfig配置类

```
package com.langfeiyes.exp.job.config;


import com.langfeiyes.exp.domain.Employee;
import com.langfeiyes.exp.job.listener.CsvToDBJobListener;
import org.apache.ibatis.session.SqlSessionFactory;
import org.mybatis.spring.batch.MyBatisBatchItemWriter;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
```

```java
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.launch.support.RunIdIncrementer;
import org.springframework.batch.item.file.FlatFileItemReader;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.PathResource;
import org.springframework.core.task.SimpleAsyncTaskExecutor;

import java.io.File;

/**
 * 将数据从csv文件中读取，并写入数据库
 */
@Configuration
public class CsvToDBJobConfig {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Autowired
    private SqlSessionFactory sqlSessionFactory;

    @Value("${job.data.path}")
    private String path;

    //多线程读-读文件，使用FlatFileItemReader
    @Bean
    public FlatFileItemReader<Employee> cvsToDBItemReader(){
        FlatFileItemReader<Employee> reader = new
FlatFileItemReaderBuilder<Employee>()
                .name("employeeCSVItemReader")
                .saveState(false) //防止状态被覆盖
                .resource(new PathResource(new File(path,
"employee.csv").getAbsolutePath()))
                .delimited()
                .names("id", "name", "age", "sex")
                .targetType(Employee.class)
                .build();

        return reader;
    }

    //数据库写-使用mybatis提供批处理读入
    @Bean
    public MyBatisBatchItemWriter<Employee> cvsToDBItemWriter(){
        MyBatisBatchItemWriter<Employee> itemWriter = new
MyBatisBatchItemWriter<>();
        itemWriter.setSqlSessionFactory(sqlSessionFactory); //需要指定sqlsession工
厂
        //指定要操作sql语句，路径id为：EmployeeMapper.xml定义的sql语句id
```

```
  itemWriter.setStatementId("com.langfeiyes.exp.mapper.EmployeeMapper.saveTemp");
  //操作sql
        return itemWriter;
    }

    @Bean
    public Step csvToDBStep(){
        return stepBuilderFactory.get("csvToDBStep")
                .<Employee, Employee>chunk(10000)  //每个块10000个 共50个
                .reader(cvsToDBItemReader())
                .writer(cvsToDBItemWriter())
                .taskExecutor(new SimpleAsyncTaskExecutor())  //多线程读写
                .build();

    }

    //job监听器
    @Bean
    public CsvToDBJobListener csvToDBJobListener(){
        return new CsvToDBJobListener();
    }

    @Bean
    public Job csvToDBJob(){
        return jobBuilderFactory.get("csvToDB-step-job")
                .start(csvToDBStep())
                .incrementer(new RunIdIncrementer()) //保证可以多次执行
                .listener(csvToDBJobListener())
                .build();

    }
}
```

步骤7：在com.langfeiyes.exp.controller 添加JobController

```
package com.langfeiyes.exp.controller;

import com.langfeiyes.exp.service.IEmployeeService;
import org.springframework.batch.core.*;
import org.springframework.batch.core.explore.JobExplorer;
import org.springframework.batch.core.launch.JobLauncher;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.Date;

@RestController
public class JobController {

    @Autowired
    private IEmployeeService employeeService;

    @Autowired
    private JobLauncher jobLauncher;

    @Autowired
    private JobExplorer jobExplorer;

    @Autowired
    @Qualifier("csvToDBJob")
    private Job csvToDBJob;

    @GetMapping("/csvToDB")
    public String csvToDB() throws Exception {
        employeeService.truncateTemp(); //清空数据运行多次执行

        //需要多次执行，run.id 必须重写之前，再重构一个新的参数对象
        JobParameters jobParameters = new JobParametersBuilder(new
JobParameters(),jobExplorer)
                .addLong("time", new Date().getTime())
                .getNextJobParameters(csvToDBJob).toJobParameters();
        JobExecution run = jobLauncher.run(csvToDBJob, jobParameters);
        return run.getId().toString();
    }
}
```

步骤8：访问测试：http://localhost:8080/csvToDB

```
-------------------------【CsvToDBJob开始时间：】---->1670575356773<---------------
--------------
-------------------------【CsvToDBJob结束时间：】---->1670575510967<---------------
--------------
-------------------------【CsvToDBJob总耗时：】---->154194<------------------------
-----
```

13.6 需求三

需求：使用分区的方式将employee_temp表的数据读取并写入到employee表

步骤1：在com.langfeiyes.exp.job.config 包添加DBToDBJobConfig， 配置从数据库到数据库的作业

```
package com.langfeiyes.exp.job.config;


import com.langfeiyes.exp.domain.Employee;
import com.langfeiyes.exp.job.partitioner.DBToDBPartitioner;
import org.apache.ibatis.session.SqlSessionFactory;
import org.mybatis.spring.batch.MyBatisBatchItemWriter;
import org.mybatis.spring.batch.MyBatisPagingItemReader;
import org.springframework.batch.core.Job;
import org.springframework.batch.core.Step;
import
org.springframework.batch.core.configuration.annotation.JobBuilderFactory;
import
org.springframework.batch.core.configuration.annotation.StepBuilderFactory;
import org.springframework.batch.core.configuration.annotation.StepScope;
import org.springframework.batch.core.launch.support.RunIdIncrementer;
import org.springframework.batch.core.partition.PartitionHandler;
```

```
import
org.springframework.batch.core.partition.support.TaskExecutorPartitionHandler;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.task.SimpleAsyncTaskExecutor;

import java.io.File;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * 将数据从employee_temp中读取，并写入employe 表
 */
@Configuration
public class DBToDBJobConfig {
    @Autowired
    private JobBuilderFactory jobBuilderFactory;
    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Autowired
    private SqlSessionFactory sqlSessionFactory;

    public static int PAGESIZE = 1000;    //mybatis分页读取数据，跟chunkSize 一样
    public static int RANGE = 10000;   //每个分区读取数据范围(理解为个数)
    public static int GRIDSIZE = 50;   //分区个数


    //读数据-从employee_temp 表读 -- mybatis
    @Bean
    @StepScope
    public MyBatisPagingItemReader<Employee> dBToDBJobItemReader(
            @Value("#{stepExecutionContext[from]}") final Integer from,
            @Value("#{stepExecutionContext[to]}") final Integer to,
            @Value("#{stepExecutionContext[range]}") final Integer range){

        System.out.println("----------MyBatisPagingItemReader开始-----from: " +
from + "  -----to:" + to + "  -----每片数量:" + range);
        MyBatisPagingItemReader<Employee> itemReader = new
MyBatisPagingItemReader<Employee>();
        itemReader.setSqlSessionFactory(sqlSessionFactory);

 itemReader.setQueryId("com.langfeiyes.exp.mapper.EmployeeMapper.selectTempForLi
st");
        itemReader.setPageSize(DBToDBJobConfig.PAGESIZE);
        Map<String, Object> map = new HashMap<>();
        map.put("from", from);
        map.put("to", to);
        itemReader.setParameterValues(map);

        return itemReader;
    }


    //数据库写- 写入到employee 表中
```

```
   @Bean
    public MyBatisBatchItemWriter<Employee> dbToDBItemWriter(){
        MyBatisBatchItemWriter<Employee> itemWriter = new
MyBatisBatchItemWriter<>();
        itemWriter.setSqlSessionFactory(sqlSessionFactory);

 itemWriter.setStatementId("com.langfeiyes.exp.mapper.EmployeeMapper.save");  //
操作sql
        return itemWriter;
    }

    //文件分区处理器-处理分区
    @Bean
    public PartitionHandler dbToDBPartitionHandler() {
        TaskExecutorPartitionHandler handler = new
TaskExecutorPartitionHandler();
        handler.setGridSize(DBToDBJobConfig.GRIDSIZE);
        handler.setTaskExecutor(new SimpleAsyncTaskExecutor());
        handler.setStep(workStep());
        try {
            handler.afterPropertiesSet();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return handler;
    }

    //每个从分区操作步骤
    @Bean
    public Step workStep() {
        return stepBuilderFactory.get("workStep")
                .<Employee, Employee>chunk(DBToDBJobConfig.PAGESIZE)
                .reader(dBToDBJobItemReader(null, null, null))
                .writer(dbToDBItemWriter())
                .build();
    }

    @Bean
    public DBToDBPartitioner dbToDBPartitioner(){
        return new DBToDBPartitioner();
    }

    //主分区操作步骤
    @Bean
    public Step masterStep() {
        return stepBuilderFactory.get("masterStep")
                .partitioner(workStep().getName(),dbToDBPartitioner())
                .partitionHandler(dbToDBPartitionHandler())
                .build();
    }

    @Bean
    public Job dbToDBJob(){
        return jobBuilderFactory.get("dbToDB-step-job")
                .start(masterStep())
                .incrementer(new RunIdIncrementer())
                .build();
    }
```

```
}
```

步骤2：修改EmployeeMapper.xml

```
<select id="selectTempForList" resultMap="BaseResultMap">
    select * from employee_temp where id between #{from} and #{to}  limit #
{_pagesize} OFFSET #{_skiprows}
</select>
```

步骤3：在com.langfeiyes.exp.job.partitioner 创建DBToDBPartitioner， 用于分区

```
package com.langfeiyes.exp.job.partitioner;

import com.langfeiyes.exp.job.config.DBToDBJobConfig;
import org.springframework.batch.core.partition.support.Partitioner;
import org.springframework.batch.item.ExecutionContext;

import java.util.HashMap;
import java.util.Map;

public class DBToDBPartitioner implements Partitioner {
    //约定分50个区， 每个区10000个数据
    @Override
    public Map<String, ExecutionContext> partition(int gridSize) {
        String text = "----DBToDBPartitioner---第%s分区-----开始：%s---结束：%s---数
据量：%s--------------";

        Map<String, ExecutionContext> map = new HashMap<>();
        int from = 1;
        int to = DBToDBJobConfig.RANGE;
        int range = DBToDBJobConfig.RANGE;

        for (int i = 0; i < gridSize; i++) {
            System.out.println(String.format(text, i, from, to, (to - from +
1)));
            ExecutionContext context = new ExecutionContext();
            context.putInt("from", from);
            context.putInt("to", to);
            context.putInt("range", range);

            from += range;
            to += range;

            map.put("partition_" + i, context);
        }
        return map;
    }
}
```

步骤4：修改JobController 类

```
@GetMapping("/dbToDB")
public String dbToDB() throws Exception {
    employeeService.truncateAll();
    JobParameters jobParameters = new JobParametersBuilder(new
JobParameters(),jobExplorer)
        .addLong("time", new Date().getTime())
        .getNextJobParameters(dbToDBJob).toJobParameters();
    JobExecution run = jobLauncher.run(dbToDBJob, jobParameters);
    return run.getId().toString();
}
```

步骤8：访问：http://localhost:8080/dbToDB

```
----DBToDBPartitioner---第0分区-----开始：1---结束：10000---数据量：10000------------
--
----DBToDBPartitioner---第1分区-----开始：10001---结束：20000---数据量：10000--------
------
----DBToDBPartitioner---第2分区-----开始：20001---结束：30000---数据量：10000--------
------
----DBToDBPartitioner---第3分区-----开始：30001---结束：40000---数据量：10000--------
------
----DBToDBPartitioner---第4分区-----开始：40001---结束：50000---数据量：10000--------
------
----DBToDBPartitioner---第5分区-----开始：50001---结束：60000---数据量：10000--------
------
----DBToDBPartitioner---第6分区-----开始：60001---结束：70000---数据量：10000--------
------
----DBToDBPartitioner---第7分区-----开始：70001---结束：80000---数据量：10000--------
------
----DBToDBPartitioner---第8分区-----开始：80001---结束：90000---数据量：10000--------
------
----DBToDBPartitioner---第9分区-----开始：90001---结束：100000---数据量：10000-------
-------
----DBToDBPartitioner---第10分区-----开始：100001---结束：110000---数据量：10000-----
---------
----DBToDBPartitioner---第11分区-----开始：110001---结束：120000---数据量：10000-----
---------
----DBToDBPartitioner---第12分区-----开始：120001---结束：130000---数据量：10000-----
---------
----DBToDBPartitioner---第13分区-----开始：130001---结束：140000---数据量：10000-----
---------
----DBToDBPartitioner---第14分区-----开始：140001---结束：150000---数据量：10000-----
---------
----DBToDBPartitioner---第15分区-----开始：150001---结束：160000---数据量：10000-----
---------
----DBToDBPartitioner---第16分区-----开始：160001---结束：170000---数据量：10000-----
---------
----DBToDBPartitioner---第17分区-----开始：170001---结束：180000---数据量：10000-----
---------
----DBToDBPartitioner---第18分区-----开始：180001---结束：190000---数据量：10000-----
---------
----DBToDBPartitioner---第19分区-----开始：190001---结束：200000---数据量：10000-----
---------
----DBToDBPartitioner---第20分区-----开始：200001---结束：210000---数据量：10000-----
---------
```

```
----DBToDBPartitioner---第21分区-----开始：210001---结束：220000---数据量：10000-----
---------
----DBToDBPartitioner---第22分区-----开始：220001---结束：230000---数据量：10000-----
---------
----DBToDBPartitioner---第23分区-----开始：230001---结束：240000---数据量：10000-----
---------
----DBToDBPartitioner---第24分区-----开始：240001---结束：250000---数据量：10000-----
---------
----DBToDBPartitioner---第25分区-----开始：250001---结束：260000---数据量：10000-----
---------
----DBToDBPartitioner---第26分区-----开始：260001---结束：270000---数据量：10000-----
---------
----DBToDBPartitioner---第27分区-----开始：270001---结束：280000---数据量：10000-----
---------
----DBToDBPartitioner---第28分区-----开始：280001---结束：290000---数据量：10000-----
---------
----DBToDBPartitioner---第29分区-----开始：290001---结束：300000---数据量：10000-----
---------
----DBToDBPartitioner---第30分区-----开始：300001---结束：310000---数据量：10000-----
---------
----DBToDBPartitioner---第31分区-----开始：310001---结束：320000---数据量：10000-----
---------
----DBToDBPartitioner---第32分区-----开始：320001---结束：330000---数据量：10000-----
---------
----DBToDBPartitioner---第33分区-----开始：330001---结束：340000---数据量：10000-----
---------
----DBToDBPartitioner---第34分区-----开始：340001---结束：350000---数据量：10000-----
---------
----DBToDBPartitioner---第35分区-----开始：350001---结束：360000---数据量：10000-----
---------
----DBToDBPartitioner---第36分区-----开始：360001---结束：370000---数据量：10000-----
---------
----DBToDBPartitioner---第37分区-----开始：370001---结束：380000---数据量：10000-----
---------
----DBToDBPartitioner---第38分区-----开始：380001---结束：390000---数据量：10000-----
---------
----DBToDBPartitioner---第39分区-----开始：390001---结束：400000---数据量：10000-----
---------
----DBToDBPartitioner---第40分区-----开始：400001---结束：410000---数据量：10000-----
---------
----DBToDBPartitioner---第41分区-----开始：410001---结束：420000---数据量：10000-----
---------
----DBToDBPartitioner---第42分区-----开始：420001---结束：430000---数据量：10000-----
---------
----DBToDBPartitioner---第43分区-----开始：430001---结束：440000---数据量：10000-----
---------
----DBToDBPartitioner---第44分区-----开始：440001---结束：450000---数据量：10000-----
---------
----DBToDBPartitioner---第45分区-----开始：450001---结束：460000---数据量：10000-----
---------
----DBToDBPartitioner---第46分区-----开始：460001---结束：470000---数据量：10000-----
---------
----DBToDBPartitioner---第47分区-----开始：470001---结束：480000---数据量：10000-----
---------
----DBToDBPartitioner---第48分区-----开始：480001---结束：490000---数据量：10000-----
---------
----DBToDBPartitioner---第49分区-----开始：490001---结束：500000---数据量：10000-----
---------
```

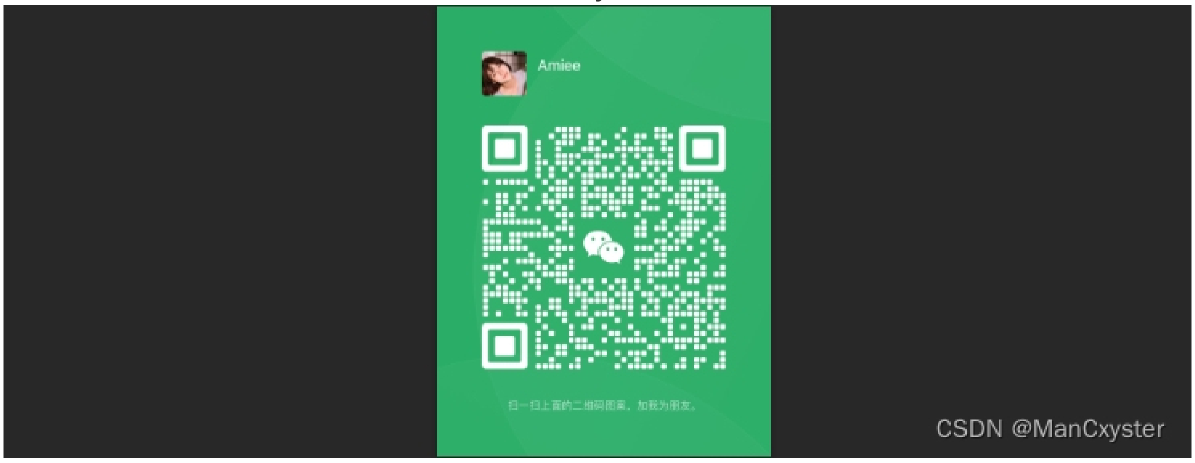
```
----------MyBatisPagingItemReader开始-----from: 250001  -----to:260000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 290001  -----to:300000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 80001  -----to:90000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 410001  -----to:420000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 360001  -----to:370000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 230001  -----to:240000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 40001  -----to:50000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 340001  -----to:350000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 450001  -----to:460000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 110001  -----to:120000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 350001  -----to:360000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 50001  -----to:60000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 430001  -----to:440000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 20001  -----to:30000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 120001  -----to:130000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 190001  -----to:200000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 100001  -----to:110000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 470001  -----to:480000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 60001  -----to:70000  -----每片数量:10000
----------MyBatisPagingItemReader开始-----from: 200001  -----to:210000  -----每片数量:10000
```

到这，案例就全部结束了。

ps:

一. 需要文档的同学点击获取↓↓↓:
Spring Batch批处理详解资料
二.视频学习Spring Batch批处理↓↓↓:
SpringBatch高效批处理框架详解及实战演练(深入浅出,全程干货)

三. 进学习交流群,免费领取学习文档,扫码或搜索:May793518,添加wx:



创作不易, 转载请注明出处!