

Spring Boot + Spring Batch 实现批处理任务，保姆级教程！（场景实战）

 cnblogs.com/javastack/p/17653257.html

来源：blog.csdn.net/qq_35387940/article/details/108193473

前言

概念词就不多说了，我简单地介绍下，spring batch 是一个方便使用的较健全的批处理框架。

为什么说是方便使用的，因为这是基于spring的一个框架，接入简单、易理解、流程分明。

为什么说是较健全的，因为它提供了往常我们在对大批量数据进行处理时需要考虑到的日志跟踪、事务粒度调配、可控执行、失败机制、重试机制、数据读写等。

正文

那么回到文章，我们该篇文章将会带来给大家的是什么？（结合实例讲解那是当然的）

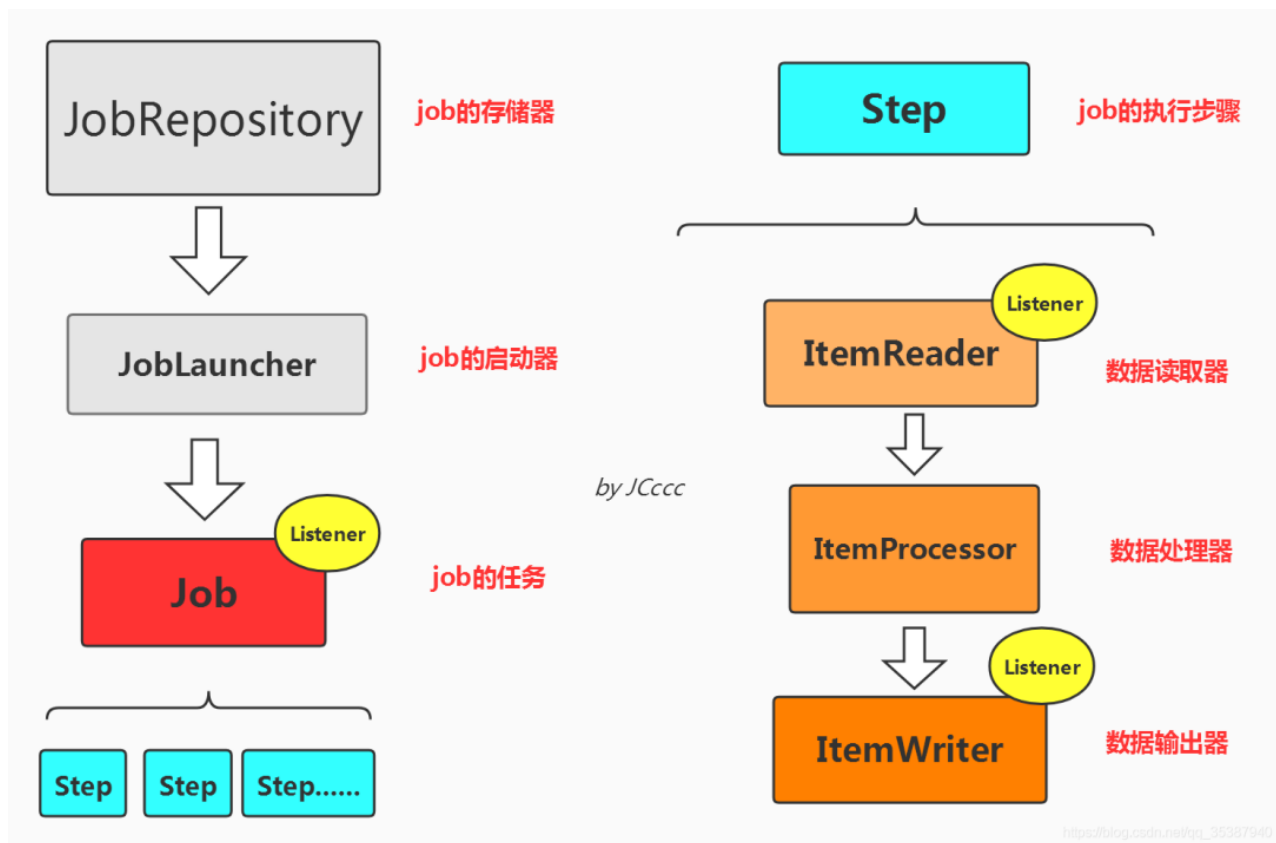
从实现的业务场景来说，有以下两个：

1. 从 csv文件 读取数据，进行业务处理再存储
2. 从 数据库 读取数据，进行业务处理再存储

也就是平时经常遇到的数据清理或者数据过滤，又或者是数据迁移备份等等。大批量的数据，自己实现分批处理需要考虑的东西太多了，又不放心，那么使用 Spring Batch 框架是一个很好的选择。

首先，在进入实例教程前，我们看看这次的实例里，我们使用springboot 整合spring batch 框架，要编码的东西有什么？

通过一张简单的图来了解：




可能大家看到这个图，是不是多多少少想起来定时任务框架？确实有那么点像，但是我必须在这告诉大家，这是一个批处理框架，不是一个schuedling 框架。但是前面提到它提供了可执行控制，也就是说，啥时候执行是可控的，那么显然就是自己可以进行扩展结合定时任务框架，实现你心中所想。

ok，回到主题，相信大家能从图中简单明了地看到我们这次实例，需要实现的东西有什么了。所以我就不会对各个小组件进行大批量文字的描述了。

那么我们事不宜迟，开始我们的实例教程。

首先准备一个数据库，里面建一张简单的表，用于实例数据的写入存储或者说是读取等等。

bloginfo表

| 字段 | 索引 | 外键 | 触发器 | 选项 | 注释 | SQL 预览 | | | | | |
|------------|----|----|-----|----|---------|--------|-----|-------------------------------------|--------------------------|---|--------|
| 名 | | | | | 类型 | 长度 | 小数点 | 不是 null | 虚拟 | 键 | 注释 |
| ▶ id | | | | | int | 11 | 0 | <input checked="" type="checkbox"/> | <input type="checkbox"/> |  1 | 主键 |
| blogAuthor | | | | | varchar | 255 | 0 | <input type="checkbox"/> | <input type="checkbox"/> | | 博客作者标识 |
| blogUrl | | | | | varchar | 255 | 0 | <input type="checkbox"/> | <input type="checkbox"/> | | 博客链接 |
| blogTitle | | | | | varchar | 255 | 0 | <input type="checkbox"/> | <input type="checkbox"/> | | 博客标题 |
| blogItem | | | | | varchar | 255 | 0 | <input type="checkbox"/> | <input type="checkbox"/> | | 博客栏目 |

bloginfo 表

https://blog.csdn.net/qq_35387940

相关建表sql语句：

```
CREATE TABLE `bloginfo` (  
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '主键',  
  `blogAuthor` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL  
DEFAULT NULL COMMENT '博客作者标识',  
  `blogUrl` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT  
NULL COMMENT '博客链接',  
  `blogTitle` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT  
NULL COMMENT '博客标题',  
  `blogItem` varchar(255) CHARACTER SET utf8 COLLATE utf8_general_ci NULL DEFAULT  
NULL COMMENT '博客栏目',  
  PRIMARY KEY (`id`) USING BTREE  
) ENGINE = InnoDB AUTO_INCREMENT = 89031 CHARACTER SET = utf8 COLLATE =  
utf8_general_ci ROW_FORMAT = Dynamic;
```

pom文件里的核心依赖：

```

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

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

<!-- spring batch -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-batch</artifactId>
</dependency>

<!-- hibernate validator -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.7.Final</version>
</dependency>

<!-- mybatis -->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.0.0</version>
</dependency>

<!-- mysql -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>

<!-- druid数据源驱动 1.1.10解决springboot从1.0—2.0版本问题 -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.18</version>
</dependency>

```

yml文件：

Spring Boot 基础就不介绍了，推荐看这个实战项目：

| <https://github.com/javastacks/spring-boot-best-practice>

```

spring:
  batch:
    job:
      #设置为 false -需要jobLauncher.run执行
      enabled: false
      initialize-schema: always
      # table-prefix: my-batch

datasource:
  druid:
    username: root
    password: root
    url: jdbc:mysql://localhost:3306/hellodemo?
useSSL=false&useUnicode=true&characterEncoding=UTF-
8&serverTimezone=GMT%2B8&zeroDateTimeBehavior=convertToNull
    driver-class-name: com.mysql.cj.jdbc.Driver
    initialSize: 5
    minIdle: 5
    maxActive: 20
    maxWait: 60000
    timeBetweenEvictionRunsMillis: 60000
    minEvictableIdleTimeMillis: 300000
    validationQuery: SELECT 1 FROM DUAL
    testWhileIdle: true
    testOnBorrow: false
    testOnReturn: false
    poolPreparedStatements: true
    maxPoolPreparedStatementPerConnectionSize: 20
    useGlobalDataSourceStat: true
    connectionProperties: druid.stat.mergeSql=true;druid.stat.slowSqlMillis=5000
server:
  port: 8665

```



ps：这里我们用到了druid数据库连接池，其实有个小坑，后面文章会讲到。

因为我们这次的实例最终数据处理完之后，是写入数据库存储（当然你也可以输出到文件等等）。

所以我们前面也建了一张表，pom文件里面我们也整合的mybatis，那么我们在整合spring batch 主要编码前，我们先把这些关于数据库打通用到的简单过一下。

pojo 层

BlogInfo.java：

```

/**
 * @Author : JCccc
 * @Description :
 */
public class BlogInfo {

    private Integer id;
    private String blogAuthor;
    private String blogUrl;
    private String blogTitle;
    private String blogItem;

    @Override
    public String toString() {
        return "BlogInfo{" +
            "id=" + id +
            ", blogAuthor='" + blogAuthor + '\'' +
            ", blogUrl='" + blogUrl + '\'' +
            ", blogTitle='" + blogTitle + '\'' +
            ", blogItem='" + blogItem + '\'' +
            '}';
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getBlogAuthor() {
        return blogAuthor;
    }

    public void setBlogAuthor(String blogAuthor) {
        this.blogAuthor = blogAuthor;
    }

    public String getBlogUrl() {
        return blogUrl;
    }

    public void setBlogUrl(String blogUrl) {
        this.blogUrl = blogUrl;
    }

    public String getBlogTitle() {
        return blogTitle;
    }

    public void setBlogTitle(String blogTitle) {
        this.blogTitle = blogTitle;
    }

    public String getBlogItem() {

```

```

        return blogItem;
    }

    public void setBlogItem(String blogItem) {
        this.blogItem = blogItem;
    }
}

```

mapper层

BlogMapper.java :

ps：可以看到这个实例我用的是注解的方式，哈哈为了省事，而且我还不写service层和impl层，也是为了省事，因为该篇文章重点不在这些，所以这些不好的大家不要学。

```

import com.example.batchdemo.pojo.BlogInfo;
import org.apache.ibatis.annotations.*;
import java.util.List;
import java.util.Map;

/**
 * @Author : JCccc
 * @Description :
 */
@Mapper
public interface BlogMapper {
    @Insert("INSERT INTO bloginfo ( blogAuthor, blogUrl, blogTitle, blogItem )
VALUES ( #{blogAuthor}, #{blogUrl},#{blogTitle},#{blogItem}) ")
    @Options(useGeneratedKeys = true, keyProperty = "id")
    int insert(BlogInfo bloginfo);

    @Select("select blogAuthor, blogUrl, blogTitle, blogItem from bloginfo where
blogAuthor < #{authorId}")
    List<BlogInfo> queryInfoById(Map<String , Integer> map);
}

```

接下来，重头戏，我们开始对前边那张图里涉及到的各个小组件进行编码。

首先创建一个配置类，MyBatchConfig.java：

从我起名来看，可以知道这基本就是咱们整合spring batch 涉及到的一些配置组件都会写在这里了。

首先我们按照咱们上面的图来看，里面包含内容有：

```

JobRepository job的注册/存储器
JobLauncher job的执行器
Job job任务，包含一个或多个Step
Step 包含 (ItemReader、ItemProcessor和ItemWriter)
ItemReader 数据读取器
ItemProcessor 数据处理器
ItemWriter 数据输出器

```

首先，在MyBatchConfig类前加入注解：

`@Configuration` 用于告诉spring，咱们这个类是一个自定义配置类，里面很多bean都需要加载到spring容器里面

`@EnableBatchProcessing` 开启批处理支持

```
@Configuration
@EnableBatchProcessing // 开启批处理的支持
public class MyBatchConfig {
    private Logger logger = LoggerFactory.getLogger(MyBatchConfig.class);
```

然后开始往MyBatchConfig类里，编写各个小组件。

JobRepository

写在MyBatchConfig类里

```
/**
 * JobRepository定义：Job的注册容器以及和数据库打交道（事务管理等）
 * @param dataSource
 * @param transactionManager
 * @return
 * @throws Exception
 */
@Bean
public JobRepository myJobRepository(DataSource dataSource,
PlatformTransactionManager transactionManager) throws Exception{
    JobRepositoryFactoryBean jobRepositoryFactoryBean = new
JobRepositoryFactoryBean();
    jobRepositoryFactoryBean.setDatabaseType("mysql");
    jobRepositoryFactoryBean.setTransactionManager(transactionManager);
    jobRepositoryFactoryBean.setDataSource(dataSource);
    return jobRepositoryFactoryBean.getObject();
}
```

JobLauncher

写在MyBatchConfig类里

```
/**
 * jobLauncher定义：job的启动器,绑定相关的jobRepository
 * @param dataSource
 * @param transactionManager
 * @return
 * @throws Exception
 */
@Bean
public SimpleJobLauncher myJobLauncher(DataSource dataSource,
PlatformTransactionManager transactionManager) throws Exception{
    SimpleJobLauncher jobLauncher = new SimpleJobLauncher();
    // 设置jobRepository
    jobLauncher.setJobRepository(myJobRepository(dataSource, transactionManager));
    return jobLauncher;
}
```


Job

写在MyBatchConfig类里

```
/**
 * 定义job
 * @param jobs
 * @param myStep
 * @return
 */
@Bean
public Job myJob(JobBuilderFactory jobs, Step myStep){
    return jobs.get("myJob")
        .incrementer(new RunIdIncrementer())
        .flow(myStep)
        .end()
        .listener(myJobListener())
        .build();
}
```

对于Job的运行，是可以配置监听器的

JobListener

写在MyBatchConfig类里

```
/**
 * 注册job监听器
 * @return
 */
@Bean
public MyJobListener myJobListener(){
    return new MyJobListener();
}
```

这是一个我们自己自定义的监听器，所以是单独创建的，[MyJobListener.java](#)：

```
/**
 * @Author : JCccc
 * @Description :监听Job执行情况，实现JobExecutionListener，且在batch配置类里，Job的Bean
上绑定该监听器
 */

public class MyJobListener implements JobExecutionListener {

    private Logger logger = LoggerFactory.getLogger(MyJobListener.class);

    @Override
    public void beforeJob(JobExecution jobExecution) {
        logger.info("job 开始, id={}", jobExecution.getJobId());
    }

    @Override
    public void afterJob(JobExecution jobExecution) {
        logger.info("job 结束, id={}", jobExecution.getJobId());
    }
}
```

Step (ItemReader ItemProcessor ItemWriter)

step里面包含数据读取器，数据处理器，数据输出器三个小组件的的实现。

我们也是一个个拆解来进行编写。

文章前边说到，该篇实现的场景包含两种，一种是从csv文件读入大量数据进行处理，另一种是从数据库表读入大量数据进行处理。

从CSV文件读取数据

ItemReader

写在MyBatchConfig类里

```
/**
 * ItemReader定义：读取文件数据+entity实体类映射
 * @return
 */
@Bean
public ItemReader<BlogInfo> reader(){
    // 使用FlatFileItemReader去读csv文件，一行即一条数据
    FlatFileItemReader<BlogInfo> reader = new FlatFileItemReader<>();
    // 设置文件处在路径
    reader.setResource(new ClassPathResource("static/bloginfo.csv"));
    // entity与csv数据做映射
    reader.setLineMapper(new DefaultLineMapper<BlogInfo>() {
        {
            setLineTokenizer(new DelimitedLineTokenizer() {
                {
                    setNames(new String[]
{"blogAuthor", "blogUrl", "blogTitle", "blogItem"});
                }
            });
            setFieldSetMapper(new BeanWrapperFieldSetMapper<BlogInfo>() {
                {
                    setTargetType(BlogInfo.class);
                }
            });
        }
    });
    return reader;
}
```

简单代码解析：

```

/**
 * ItemReader 定义：读取文件数据, 实体类映射
 * @return
 */
@Bean
public ItemReader<BlogInfo> reader(){
    // 使用FlatFileItemReader去读csv文件，一行即一条数据
    FlatFileItemReader<BlogInfo> reader = new FlatFileItemReader<>();
    // 设置文件处在路径
    reader.setResource(new ClassPathResource("static/bloginfo.csv")); 数据读取的来源csv 文件
    // entity与csv数据做映射
    reader.setLineMapper(new DefaultLineMapper<BlogInfo>() {
        {
            setLineTokenizer(new DelimitedLineTokenizer() {
                {
                    setNames(new String[]{"blogAuthor","blogUrl","blogTitle","blogItem"});
                }
            });
            setFieldSetMapper(new BeanWrapperFieldSetMapper<BlogInfo>() {
                {
                    setTargetType(BlogInfo.class);
                }
            });
        }
    });
    return reader;
}

```

https://blog.csdn.net/qq_35387940

对于数据读取器 ItemReader ，我们给它安排了一个读取监听器，创建 `MyReadListener.java`：

```

/**
 * @Author : JCccc
 * @Description :
 */

public class MyReadListener implements ItemReadListener<BlogInfo> {

    private Logger logger = LoggerFactory.getLogger(MyReadListener.class);

    @Override
    public void beforeRead() {
    }

    @Override
    public void afterRead(BlogInfo item) {
    }

    @Override
    public void onReadError(Exception ex) {
        try {
            logger.info(format("%s%n", ex.getMessage()));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

ItemProcessor

写在MyBatchConfig类里

```

/**
 * 注册ItemProcessor：处理数据+校验数据
 * @return
 */
@Bean
public ItemProcessor<BlogInfo, BlogInfo> processor(){
    MyItemProcessor myItemProcessor = new MyItemProcessor();
    // 设置校验器
    myItemProcessor.setValidator(myBeanValidator());
    return myItemProcessor;
}

```

数据处理器，是我们自定义的，里面主要是包含我们对数据处理的业务逻辑，并且我们设置了一些数据校验器，我们这里使用 JSR-303的Validator来作为校验器。

校验器

写在MyBatchConfig类里

```

/**
 * 注册校验器
 * @return
 */
@Bean
public MyBeanValidator myBeanValidator(){
    return new MyBeanValidator<BlogInfo>();
}

```

创建MyItemProcessor.java：

ps：里面我的数据处理逻辑是，获取出读取数据里面的每条数据的blogItem字段，如果是springboot，那就对title字段值进行替换。

其实也就是模拟一个简单地数据处理场景。

```

import com.example.batchdemo.pojo.BlogInfo;
import org.springframework.batch.item.validator.ValidatingItemProcessor;
import org.springframework.batch.item.validator.ValidationException;

/**
 * @Author : JCccc
 * @Description :
 */
public class MyItemProcessor extends ValidatingItemProcessor<BlogInfo> {
    @Override
    public BlogInfo process(BlogInfo item) throws ValidationException {
        /**
         * 需要执行super.process(item)才会调用自定义校验器
         */
        super.process(item);
        /**
         * 对数据进行简单的处理
         */
        if (item.getBlogItem().equals("springboot")) {
            item.setBlogTitle("springboot 系列还请看看我Jc");
        } else {
            item.setBlogTitle("未知系列");
        }
        return item;
    }
}

```

创建MyBeanValidator.java :

```

import org.springframework.batch.item.validator.ValidationException;
import org.springframework.batch.item.validator.Validator;
import org.springframework.beans.factory.InitializingBean;
import javax.validation.ConstraintViolation;
import javax.validation.Validation;
import javax.validation.ValidatorFactory;
import java.util.Set;

/**
 * @Author : JCccc
 * @Description :
 */
public class MyBeanValidator<T> implements Validator<T>, InitializingBean {

    private javax.validation.Validator validator;

    @Override
    public void validate(T value) throws ValidationException {
        /**
         * 使用Validator的validate方法校验数据
         */
        Set<ConstraintViolation<T>> constraintViolations =
            validator.validate(value);
        if (constraintViolations.size() > 0) {
            StringBuilder message = new StringBuilder();
            for (ConstraintViolation<T> constraintViolation :
constraintViolations) {
                message.append(constraintViolation.getMessage() + "\n");
            }
            throw new ValidationException(message.toString());
        }
    }

    /**
     * 使用JSR-303的Validator来校验我们的数据，在此进行JSR-303的Validator的初始化
     * @throws Exception
     */
    @Override
    public void afterPropertiesSet() throws Exception {
        ValidatorFactory validatorFactory =
            Validation.buildDefaultValidatorFactory();
        validator = validatorFactory.getContext().getValidator();
    }
}

```

ps：其实该篇文章没有使用这个数据校验器，大家想使用的话，可以在实体类上添加一些校验器的注解@NotNull @Max @Email等等。我偏向于直接在处理器里面进行处理，想把关于数据处理的代码都写在一块。

ItemWriter

写在MyBatchConfig类里

```

/**
 * ItemWriter定义：指定datasource，设置批量插入sql语句，写入数据库
 * @param dataSource
 * @return
 */
@Bean
public ItemWriter<BlogInfo> writer(DataSource dataSource){
    // 使用jdbcBatchItemWrite写数据到数据库中
    JdbcBatchItemWriter<BlogInfo> writer = new JdbcBatchItemWriter<>();
    // 设置有参数的sql语句
    writer.setItemSqlParameterSourceProvider(new
BeanPropertyItemSqlParameterSourceProvider<BlogInfo>());
    String sql = "insert into bloginfo "+" (blogAuthor,blogUrl,blogTitle,blogItem)
    "
        +" values(:blogAuthor,:blogUrl,:blogTitle,:blogItem)";
    writer.setSql(sql);
    writer.setDataSource(dataSource);
    return writer;
}

```

简单代码解析：



```

/**
 * ItemWriter定义：指定datasource，设置批量插入sql语句，写入数据库
 * @param dataSource
 * @return
 */
@Bean
public ItemWriter<BlogInfo> writer(DataSource dataSource){
    // 使用jdbcBatchItemWrite写数据到数据库中
    JdbcBatchItemWriter<BlogInfo> writer = new JdbcBatchItemWriter<>();
    // 设置有参数的sql语句
    writer.setItemSqlParameterSourceProvider(new BeanPropertyItemSqlParameterSourceProvider<BlogInfo>());
    String sql = "insert into bloginfo "+" (blogAuthor,blogUrl,blogTitle,blogItem) "
        +" values(:blogAuthor,:blogUrl,:blogTitle,:blogItem)";
    writer.setSql(sql);
    writer.setDataSource(dataSource);
    return writer;
}

```

数据源传入

插入数据sql的编写，可以看到是一个批量插入语

https://blog.csdn.net/qq_35387940

同样 对于数据读取器 ItemWriter ，我们给它也安排了一个输出监听器，创建 `MyWriteListener.java`：

```

import com.example.batchdemo.pojo.BlogInfo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.core.ItemWriteListener;
import java.util.List;
import static java.lang.String.format;

/**
 * @Author : JCccc
 * @Description :
 */
public class MyWriteListener implements ItemWriteListener<BlogInfo> {
    private Logger logger = LoggerFactory.getLogger(MyWriteListener.class);

    @Override
    public void beforeWrite(List<? extends BlogInfo> items) {
    }

    @Override
    public void afterWrite(List<? extends BlogInfo> items) {
    }

    @Override
    public void onWriteError(Exception exception, List<? extends BlogInfo> items)
    {
        try {
            logger.info(format("%s\n", exception.getMessage()));
            for (BlogInfo message : items) {
                logger.info(format("Failed writing BlogInfo : %s",
message.toString()));
            }

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

```

ItemReader、**ItemProcessor**、**ItemWriter**，这三个小组件到这里，我们都实现了，那么接下来就是把这三个小组件跟我们的step去绑定起来。

写在MyBatchConfig类里


```

/**
 * step定义 :
 * 包括
 * ItemReader 读取
 * ItemProcessor 处理
 * ItemWriter 输出
 * @param stepBuilderFactory
 * @param reader
 * @param writer
 * @param processor
 * @return
 */

@Bean
public Step myStep(StepBuilderFactory stepBuilderFactory, ItemReader<BlogInfo>
reader,
                  ItemWriter<BlogInfo> writer, ItemProcessor<BlogInfo, BlogInfo>
processor){
    return stepBuilderFactory
        .get("myStep")
        .<BlogInfo, BlogInfo>chunk(65000) // Chunk的机制(即每次读取一条数据，再处理
一条数据，累积到一定数量后再一次性交给writer进行写入操作)

        .reader(reader).faultTolerant().retryLimit(3).retry(Exception.class).skip(Exceptio
n.class).skipLimit(2)
        .listener(new MyReadListener())
        .processor(processor)
        .writer(writer).faultTolerant().skip(Exception.class).skipLimit(2)
        .listener(new MyWriteListener())
        .build();
}

```

这个Step，稍作讲解。

前边提到了，spring batch框架，提供了事务的控制，重启，检测跳过等等机制。

那么，这些东西的实现，很多都在于这个step环节的设置。

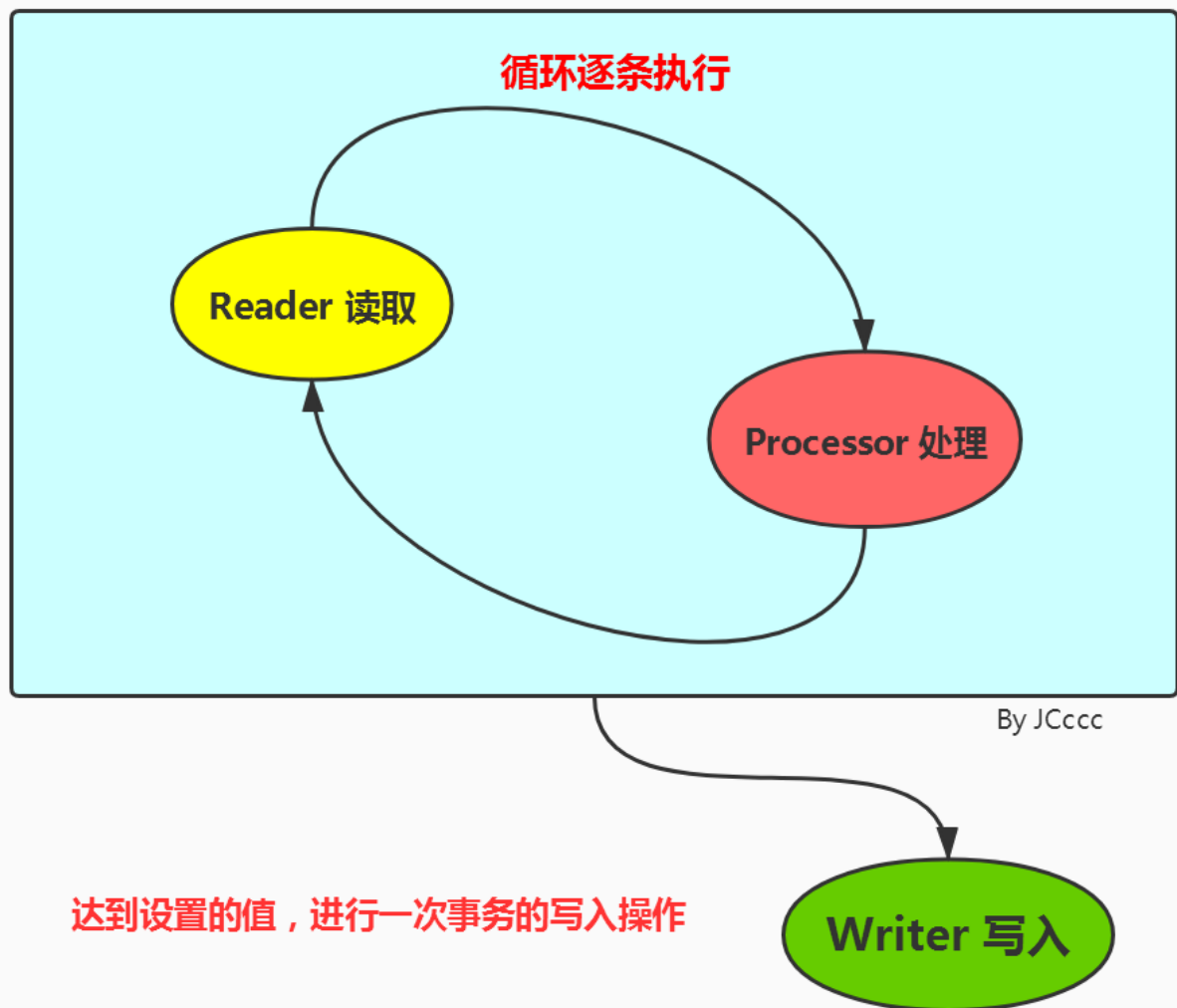
首先看到我们代码出现的第一个设置，`chunk(6500)`，Chunk的机制(即每次读取一条数据，再处理一条数据，累积到一定数量后再一次性交给writer进行写入操作。

没错，对于整个step环节，就是数据的读取，处理最后到输出。

这个chunk机制里，我们传入的 6500，也就是告诉它，读取处理数据，累计达到 6500条进行一次批次处理，去执行写入操作。

这个传值，是根据具体业务而定，可以是500条一次，1000条一次，也可以是20条一次，50条一次。

通过一张简单的小图来帮助理解：



在我们大量数据处理，不管是读取或者说是写入，都肯定会涉及到一些未知或者已知因素导致某条数据失败了。

那么如果说咱们啥也不设置，失败一条数据，那么我们就当作整个失败了？。显然这个太不人性，所以spring batch 提供了 retry 和 skip 两个设置（其实还有restart），通过这两个设置来人性化地解决一些数据操作失败场景。

```
retryLimit(3).retry(Exception.class)
```

没错，这个就是设置重试，当出现异常的时候，重试多少次。我们设置为3，也就是说当一条数据操作失败，那我们会对这条数据进行重试3次，还是失败就是 当做失败了，那么我们如果有配置skip（推荐配置使用），那么这个数据失败记录就会留到给 skip 来处理。

```
skip(Exception.class).skipLimit(2)
```

skip，跳过，也就是说我们如果设置3，那么就是可以容忍 3条数据的失败。只有达到失败数据达到3次，我们才中断这个step。

对于失败的数据，我们做了相关的监听器以及异常信息记录，供与后续手动补救。

那么记下来我们开始去调用这个批处理job，我们通过接口去触发这个批处理事件，新建一个Controller，`TestController.java`：

```
/**
 * @Author : JCccc
 * @Description :
 */
@RestController
public class TestController {
    @Autowired
    SimpleJobLauncher jobLauncher;

    @Autowired
    Job myJob;

    @GetMapping("testJob")
    public void testJob() throws JobParametersInvalidException,
        JobExecutionAlreadyRunningException, JobRestartException,
        JobInstanceAlreadyCompleteException {
        // 后置参数：使用JobParameters中绑定参数 addLong addString 等方法
        JobParameters jobParameters = new
        JobParametersBuilder().toJobParameters();
        jobLauncher.run(myJob, jobParameters);
    }
}
```

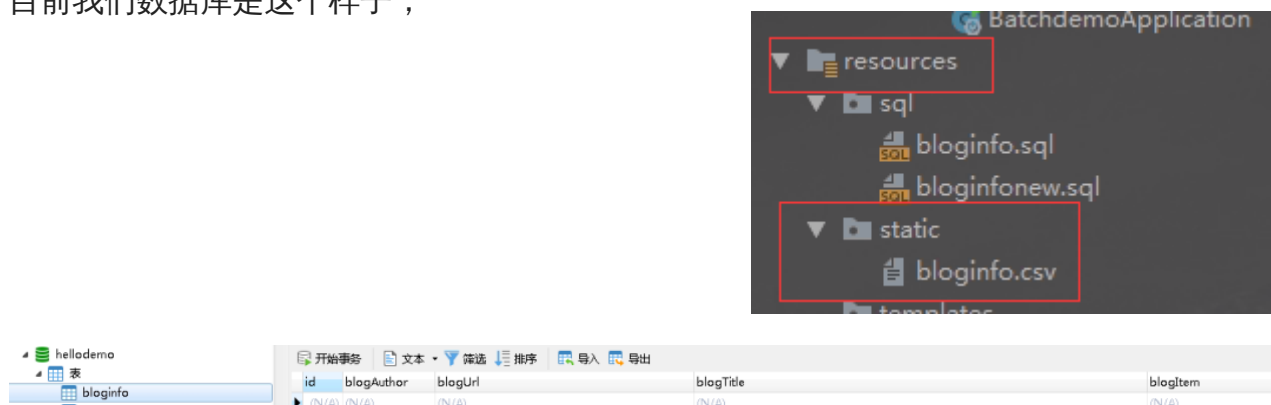
对了，我准备了一个csv文件 `bloginfo.csv`，里面大概8万多条数据，用来进行批处理测试：

| | A | B | C | D | E | F | G |
|-------|-------|------------------------|------|------------|---|---|---|
| 88792 | 10964 | http://testAddData.com | 测试数据 | springboot | | | |
| 88793 | 10965 | http://testAddData.com | 测试数据 | springboot | | | |
| 88794 | 10966 | http://testAddData.com | 测试数据 | springboot | | | |
| 88795 | 10967 | http://testAddData.com | 测试数据 | springboot | | | |
| 88796 | 10968 | http://testAddData.com | 测试数据 | springboot | | | |
| 88797 | 10969 | http://testAddData.com | 测试数据 | springboot | | | |
| 88798 | 10970 | http://testAddData.com | 测试数据 | springboot | | | |
| 88799 | 10971 | http://testAddData.com | 测试数据 | springboot | | | |
| 88800 | 10972 | http://testAddData.com | 测试数据 | springboot | | | |
| 88801 | 10973 | http://testAddData.com | 测试数据 | springboot | | | |
| 88802 | 10974 | http://testAddData.com | 测试数据 | springboot | | | |
| 88803 | 10975 | http://testAddData.com | 测试数据 | springboot | | | |
| 88804 | 10976 | http://testAddData.com | 测试数据 | springboot | | | |
| 88805 | 10977 | http://testAddData.com | 测试数据 | springboot | | | |
| 88806 | 10978 | http://testAddData.com | 测试数据 | springboot | | | |
| 88807 | 10979 | http://testAddData.com | 测试数据 | springboot | | | |
| 88808 | 10980 | http://testAddData.com | 测试数据 | springboot | | | |
| 88809 | 10981 | http://testAddData.com | 测试数据 | springboot | | | |
| 88810 | 10982 | http://testAddData.com | 测试数据 | springboot | | | |
| 88811 | 10983 | http://testAddData.com | 测试数据 | springboot | | | |
| 88812 | 10984 | http://testAddData.com | 测试数据 | springboot | | | |
| 88813 | 10985 | http://testAddData.com | 测试数据 | springboot | | | |
| 88814 | 10986 | http://testAddData.com | 测试数据 | springboot | | | |
| 88815 | 10987 | http://testAddData.com | 测试数据 | springboot | | | |
| 88816 | 10988 | http://testAddData.com | 测试数据 | springboot | | | |

这个文件的路径跟我们的数据读取器里面读取的路径要一直，

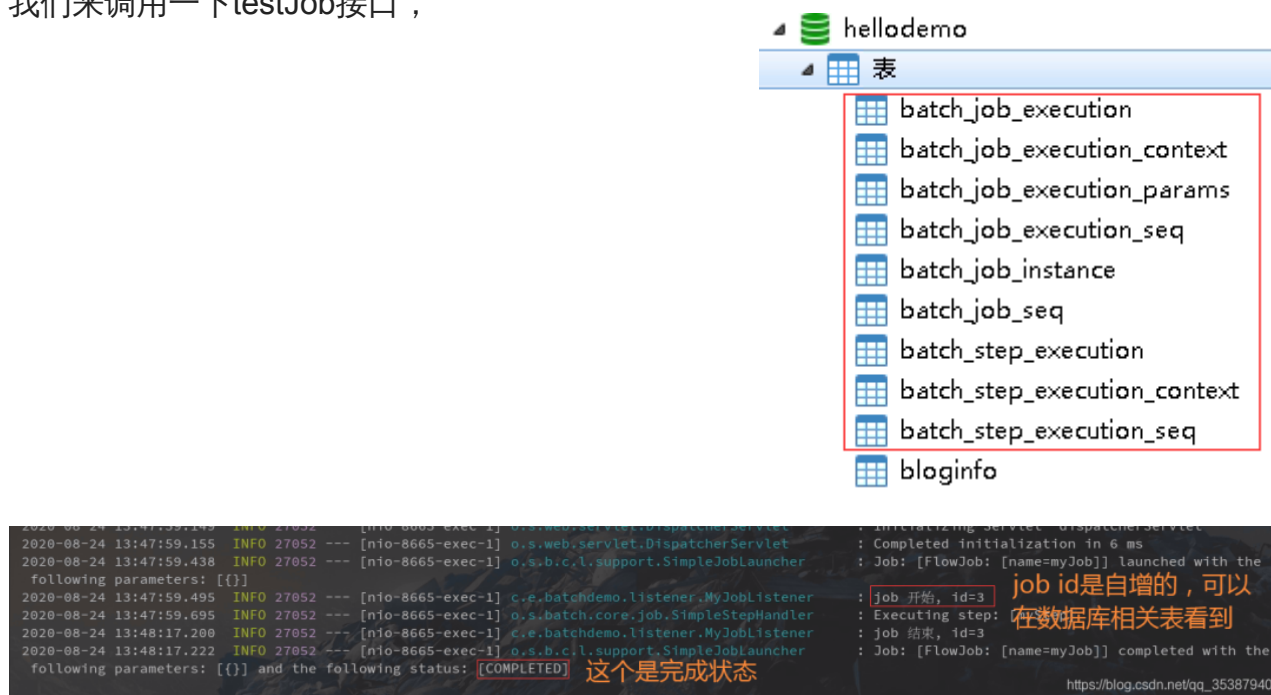
```
// 设置文件处在路径
reader.setResource(new ClassPathResource("static/bloginfo.csv"));
```

目前我们数据库是这个样子，



接下来我们把我们的项目启动起来，再看一眼数据库，生成了一些batch用来跟踪记录job的一些数据表：

我们来调用一下testJob接口，



然后看下数据库，可以看的数据全部都进行了相关的逻辑处理并插入到了数据库：

| id | blogAuthor | blogUrl | blogTitle | blogItem |
|-------|------------|------------------------|----------------------|------------|
| 88803 | 10975 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88804 | 10976 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88805 | 10977 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88806 | 10978 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88807 | 10979 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88808 | 10980 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88809 | 10981 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88810 | 10982 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88811 | 10983 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88812 | 10984 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88813 | 10985 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88814 | 10986 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88815 | 10987 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88816 | 10988 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88817 | 10989 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88818 | 10990 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88819 | 10991 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88820 | 10992 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88821 | 10993 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88822 | 10994 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88823 | 10995 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88824 | 10996 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88825 | 10997 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88826 | 10998 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |
| 88827 | 10999 | http://testAddData.com | springboot 系列还请看着我Jc | springboot |

到这里，我们对Springboot 整合 spring batch 其实已经操作完毕了，也实现了从csv文件读取数据处理存储的业务场景。

从数据库读取数据

ps：前排提示使用druid有坑。后面会讲到。

那么接下来实现场景，从数据库表内读取数据进行处理输出到新的表里面。

那么基于我们上边的整合，我们已经实现了

JobRepository job的注册/存储器
 JobLauncher job的执行器
 Job job任务，包含一个或多个Step
 Step 包含 (ItemReader、ItemProcessor和ItemWriter)
 ItemReader 数据读取器
 ItemProcessor 数据处理器
 ItemWriter 数据输出器
 job 监听器
 reader 监听器
 writer 监听器
 process 数据校验器

那么对于我们新写一个job完成 一个新的场景，我们需要全部重写么？

显然没必要，当然完全新写一套也是可以的。

那么该篇，对于一个新的也出场景，从csv文件读取数据转换到数据库表读取数据，我们重新新建的有：

1. **数据读取器**：原先使用的是 `FlatFileItemReader`，我们现在改为使用 `MyBatisCursorItemReader`
2. **数据处理器**：新的场景，业务为了好扩展，所以我们处理器最好也新建一个

3. **数据输出器**：新的场景，业务为了好扩展，所以我们数据输出器最好也新建一个
4. **step的绑定设置**：新的场景，业务为了好扩展，所以我们step最好也新建一个
5. **Job**：当然是要重新写一个了

其他我们照用原先的就行，JobRepository，JobLauncher以及各种监听器啥的，暂且不重新建了。

新建MyItemProcessorNew.java：

```
import org.springframework.batch.item.validator.ValidatingItemProcessor;
import org.springframework.batch.item.validator.ValidationException;

/**
 * @Author : JCccc
 * @Description :
 */
public class MyItemProcessorNew extends ValidatingItemProcessor<BlogInfo> {
    @Override
    public BlogInfo process(BlogInfo item) throws ValidationException {
        /**
         * 需要执行super.process(item)才会调用自定义校验器
         */
        super.process(item);
        /**
         * 对数据进行简单的处理
         */
        Integer authorId= Integer.valueOf(item.getBlogAuthor());
        if (authorId<20000) {
            item.setBlogTitle("这是都是小于20000的数据");
        } else if (authorId>20000 && authorId<30000){
            item.setBlogTitle("这是都是小于30000但是大于20000的数据");
        }else {
            item.setBlogTitle("旧书不厌百回读");
        }
        return item;
    }
}
```

然后其他重新定义的小组件，写在MyBatchConfig类里：

```

/**
 * 定义job
 * @param jobs
 * @param stepNew
 * @return
 */
@Bean
public Job myJobNew(JobBuilderFactory jobs, Step stepNew){
    return jobs.get("myJobNew")
        .incrementer(new RunIdIncrementer())
        .flow(stepNew)
        .end()
        .listener(myJobListener())
        .build();
}

@Bean
public Step stepNew(StepBuilderFactory stepBuilderFactory,
MyBatisCursorItemReader<BlogInfo> itemReaderNew,
                    ItemWriter<BlogInfo> writerNew, ItemProcessor<BlogInfo,
BlogInfo> processorNew){
    return stepBuilderFactory
        .get("stepNew")
        .<BlogInfo, BlogInfo>chunk(65000) // Chunk的机制(即每次读取一条数据，再处理
一条数据，累积到一定数量后再一次性交给writer进行写入操作)

        .reader(itemReaderNew).faultTolerant().retryLimit(3).retry(Exception.class).skip(E
xception.class).skipLimit(10)
        .listener(new MyReadListener())
        .processor(processorNew)
        .writer(writerNew).faultTolerant().skip(Exception.class).skipLimit(2)
        .listener(new MyWriteListener())
        .build();
}

@Bean
public ItemProcessor<BlogInfo, BlogInfo> processorNew(){
    MyItemProcessorNew csvItemProcessor = new MyItemProcessorNew();
    // 设置校验器
    csvItemProcessor.setValidator(myBeanValidator());
    return csvItemProcessor;
}

@Autowired
private SqlSessionFactory sqlSessionFactory;

@Bean
@StepScope
//Spring Batch提供了一个特殊的bean scope类 (StepScope:作为一个自定义的Spring bean
scope)。这个step scope的作用是连接batches的各个steps。这个机制允许配置在Spring的beans当
steps开始时才实例化并且允许你为这个step指定配置和参数。
public MyBatisCursorItemReader<BlogInfo> itemReaderNew(@Value("#
{jobParameters[authorId]}") String authorId) {

```



```

        System.out.println("开始查询数据库");

        MyBatisCursorItemReader<BlogInfo> reader = new MyBatisCursorItemReader<>
();

reader.setQueryId("com.example.batchdemo.mapper.BlogMapper.queryInfoById");

        reader.setSqlSessionFactory(sqlSessionFactory);
        Map<String , Object> map = new HashMap<>();

        map.put("authorId" , Integer.valueOf(authorId));
        reader.setParameterValues(map);
        return reader;
}

/**
 * ItemWriter定义：指定datasource，设置批量插入sql语句，写入数据库
 * @param dataSource
 * @return
 */
@Bean
public ItemWriter<BlogInfo> writerNew(DataSource dataSource){
    // 使用jdbcBatchItemWrite写数据到数据库中
    JdbcBatchItemWriter<BlogInfo> writer = new JdbcBatchItemWriter<>();
    // 设置有参数的sql语句
    writer.setItemSqlParameterSourceProvider(new
BeanPropertyItemSqlParameterSourceProvider<BlogInfo>());
    String sql = "insert into bloginfo new "+"
(blogAuthor, blogUrl, blogTitle, blogItem) "
        +" values (:blogAuthor, :blogUrl, :blogTitle, :blogItem)";
    writer.setSql(sql);
    writer.setDataSource(dataSource);
    return writer;
}

```

代码需要注意的点

数据读取器 `MyBatisCursorItemReader`

对应的mapper方法：


```
@Mapper  
public interface BlogMapper {
```

数据处理器 MyItemProcessorNew :

```
@Override  
public BlogInfo process(BlogInfo item) throws ValidationException {
```

数据输出器，新插入到别的数据库表去，特意这样为了测试：

当然我们的数据库为了测试这个场景，也是新建了一张表，bloginfo_{new} 表。

接下来，我们新写一个接口来执行新的这个job：

```

@Autowired
SimpleJobLauncher jobLauncher;

@Autowired
Job myJobNew;

@GetMapping("testJobNew")
public void testJobNew(@RequestParam("authorId") String authorId) throws
JobParametersInvalidException, JobExecutionAlreadyRunningException,
JobRestartException, JobInstanceAlreadyCompleteException {

    JobParameters jobParametersNew = new JobParametersBuilder().addLong("timeNew",
System.currentTimeMillis())
        .addString("authorId",authorId)
        .toJobParameters();
    jobLauncher.run(myJobNew,jobParametersNew);
}

```

ok，我们来调用一些这个接口：

看下控制台：

没错，这就是失败的，原因是因为跟druid有关，报了一个数据库功能不支持。这是在数据读取的时候报的错。

我初步测试认为是MyBatisCursorItemReader，druid 数据库连接池不支持。

那么，我们只需要：

注释掉druid连接池 jar依赖

yml里替换连接池配置

其实我们不配置其他连接池，springboot 2.X 版本已经为我们整合了默认的连接池HikariCP。

在Springboot2.X版本，数据库的连接池官方推荐使用HikariCP

如果不是为了druid的那些后台监控数据，sql分析等等，完全是优先使用HikariCP的。

官方的原话：

We prefer HikariCP for its performance and concurrency. If HikariCP is available, we always choose it.

翻译：

我们更喜欢hikaricpf的性能和并发性。如果有HikariCP，我们总是选择它。

所以我们就啥连接池也不配了，使用默认的HikariCP 连接池。

推荐一个开源免费的 Spring Boot 实战项目：

| <https://github.com/javastacks/spring-boot-best-practice>

当然你想配，也是可以的：

所以我们剔除掉druid链接池后，我们再来调用一下新接口：

可以看到，从数据库获取数据并进行批次处理写入job是成功的：

新的表里面插入的数据都进行了自己写的逻辑处理：

好了，springboot 整合 spring batch 批处理框架， 就到此吧。

[+加关注](#)

posted @ 2023-08-24 09:05 [Java技术栈](#) 阅读(1676) 评论(0) [编辑](#) [收藏](#) [举报](#)