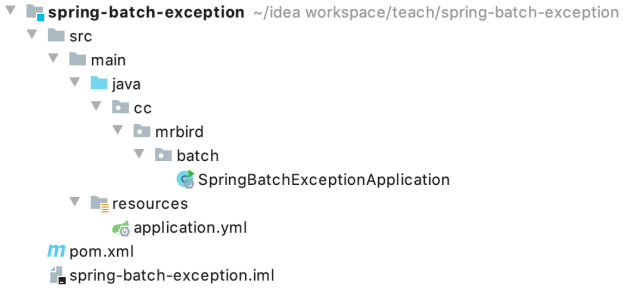
Spring Batch异常处理

mrbird.cc/Spring-Batch异常处理.html

Spring Batch处理任务过程中如果发生了异常，默认机制是马上停止任务执行，抛出相应异常，如果任务还包含未执行的步骤也不会被执行。要改变这个默认规则，我们可以配置异常重试和异常跳过机制。**异常跳过**：遇到异常的时候不希望结束任务，而是跳过这个异常，继续执行；**异常重试**：遇到异常的时候经过指定次数的重试，如果还是失败的话，才会停止任务。除了这两个特性外，本文也会记录一些别的特性。

框架搭建

新建一个Spring Boot项目，版本为2.2.4.RELEASE，artifactId为spring-batch-exception，项目结构如下图所示：



剩下的数据库层的准备，项目配置，依赖引入和Spring Batch入门文章中的框架搭建步骤一致，这里就不再赘述。

下面我们演示下，默认情况下Spring Batch处理任务遇到异常是怎么处理的。

在cc.mrbird.batch目录下新建job包，然后在该包下新建`DefaultExceptionJobDemo`：

```
@Component
public class DefaultExceptionJobDemo {

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

    @Bean
    public Job defaultExceptionJob() {
        return jobBuilderFactory.get("defaultExceptionJob")
                .start(
                    stepBuilderFactory.get("step")
                        .tasklet((stepContribution, chunkContext) -> {
                            // 获取执行上下文
                            ExecutionContext executionContext =
chunkContext.getStepContext().getStepExecution().getExecutionContext();
                            if (executionContext.containsKey("success")) {
                                System.out.println("任务执行成功");
                                return RepeatStatus.FINISHED;
                            } else {
                                String errorMessage = "处理任务过程发生异常";
                                System.out.println(errorMessage);
                                executionContext.put("success", true);
                                throw new RuntimeException(errorMessage);
                            }

                        }).build()
                ).build();
    }
}
```

上面代码中，我们在Step的`tasklet()`方法中获取了执行上下文，并且判断执行上下文中是否包含key`success`，如果包含，则任务执行成功；如果不包含，则抛出异常（抛出异常前，在执行上下文中添加`success`key）。

启动项目，控制台日志打印如下：

```
2020-03-11 17:12:50.253  INFO 38673 --- [           main] o.s.b.c.l.support.SimpleJobLauncher      : Job: [SimpleJob:
[name=defaultExceptionJob]] launched with the following parameters: [{}]
2020-03-11 17:12:50.323  INFO 38673 --- [           main] o.s.batch.core.job.SimpleStepHandler     : Executing step:
[step]
处理任务过程发生异常
2020-03-11 17:12:50.352 ERROR 38673 --- [           main] o.s.batch.core.step.AbstractStep         : Encountered an error
executing step step in job defaultExceptionJob

java.lang.RuntimeException: 处理任务过程发生异常
        at cc.mrbird.batch.job.DefaultExceptionJobDemo.lambda$defaultExceptionJob$0(DefaultExceptionJobDemo.java:38) ~
[classes/:na]
        at
org.springframework.batch.core.step.tasklet.TaskletStep$ChunkTransactionCallback.doInTransaction(TaskletStep.java:407) ~
[spring-batch-core-4.2.1.RELEASE.jar:4.2.1.RELEASE]
        at
org.springframework.batch.core.step.tasklet.TaskletStep$ChunkTransactionCallback.doInTransaction(TaskletStep.java:331) ~
[spring-batch-core-4.2.1.RELEASE.jar:4.2.1.RELEASE]
        at org.springframework.transaction.support.TransactionTemplate.execute(TransactionTemplate.java:140) ~[spring-tx-
5.2.4.RELEASE.jar:5.2.4.RELEASE]
        at org.springframework.batch.core.step.tasklet.TaskletStep$2.doInChunkContext(TaskletStep.java:273) ~[spring-
batch-core-4.2.1.RELEASE.jar:4.2.1.RELEASE]
...
```

可以看到，默认情况下，Spring Batch处理任务过程中如果发生了异常会马上停止任务的执行。

再次启动项目，控制台输出如下：

```
2020-03-11 17:14:03.184  INFO 38691 --- [           main] o.s.b.c.l.support.SimpleJobLauncher      : Job: [SimpleJob:
[name=defaultExceptionJob]] launched with the following parameters: [{}]
2020-03-11 17:14:03.264  INFO 38691 --- [           main] o.s.batch.core.job.SimpleStepHandler     : Executing step:
[step]
任务执行成功
2020-03-11 17:14:03.302  INFO 38691 --- [           main] o.s.batch.core.step.AbstractStep         : Step: [step] executed
in 37ms
2020-03-11 17:14:03.326  INFO 38691 --- [           main] o.s.b.c.l.support.SimpleJobLauncher      : Job: [SimpleJob:
[name=defaultExceptionJob]] completed with the following parameters: [{}] and the following status: [COMPLETED] in 120ms
```

因为在上次任务抛出异常前，我们在执行上下文中添加`success`key（配合MySQL持久化，不会因项目启动而丢失）。

异常重试

Spring Batch允许我们配置任务在遇到指定异常时进行指定次数的重试。在此之前，我们先定义一个自定义异常。在cc.mrbird.batch包下新建exception包，然后在该包下新建`MyJobExecutionException`：

```
public class MyJobExecutionException extends  Exception{

    private static final long serialVersionUID =
7168487913507656106L;

    public MyJobExecutionException(String message) {
        super(message);
    }
}
```

然后在job包下新建`RetryExceptionJobDemo`：

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

    @Bean
    public Job retryExceptionJob() {
        return jobBuilderFactory.get("retryExceptionJob")
                .start(step())
                .build();
    }

    private Step step() {
        return stepBuilderFactory.get("step")
                .<String, String>chunk(2)
                .reader(listItemReader())
                .processor(myProcessor())
                .writer(list -> list.forEach(System.out::println))
                .faultTolerant() // 配置错误容忍
                .retry(MyJobExecutionException.class) // 配置重试的异常类型
                .retryLimit(3) // 重试3次，三次过后还是异常的话，则任务会结束，
                // 异常的次数为reader，processor和writer中的总数，这里仅在processor里演示异
常重试
                .build();
    }

    private ListItemReader<String> listItemReader() {
        ArrayList<String> datas = new ArrayList<>();
        IntStream.range(0, 5).forEach(i -> datas.add(String.valueOf(i)));
        return new ListItemReader<>(datas);
    }

    private ItemProcessor<String, String> myProcessor() {
        return new ItemProcessor<String, String>() {
            private int count;
            @Override
            public String process(String item) throws Exception {
                System.out.println("当前处理的数据：" + item);
                if (count >= 2) {
                    return item;
                } else {
                    count++;
                    throw new MyJobExecutionException("任务处理出错");
                }
            }
        };
    }
```

在`step()`方法中，`faultTolerant()`表示开启容错功能，`retry(MyJobExecutionException.class)`表示遇到`MyJobExecutionException`异常时进行重试，`retryLimit(3)`表示如果第三次重试还是失败的话，则抛出异常，结束任务。

通过前面的学习我们知道，步骤Step包括`ItemReader`、`ItemWriter`和`ItemProcessor`，上面配置的错误容忍是针对整个Step的，所以容忍的异常次数应该是reader，processor和writer中的总数，上面的例子仅在processor里演示异常重试。

`myProcessor()`的代码逻辑很简单，就是在前两次的时候抛出`MyJobExecutionException("任务处理出错")`异常（`count < 2`），第三次的时候正常返回item（`count = 2 >= 2`），所以理论上上面的任务在重试两次之后正常运行。

启动项目，控制台打印日志如下：

```
2020-03-12 09:04:53.359  INFO 40522 --- [           main] o.s.b.c.l.support.SimpleJobLauncher      : Job: [SimpleJob:
[name=retryExceptionJob]] launched with the following parameters: [{}]
2020-03-12 09:04:53.415  INFO 40522 --- [           main] o.s.batch.core.job.SimpleStepHandler     : Executing step:
[step]
当前处理的数据：0
当前处理的数据：0
当前处理的数据：0
当前处理的数据：1
0
1
当前处理的数据：2
当前处理的数据：3
2
3
当前处理的数据：4
4
2020-03-12 09:04:53.498  INFO 40522 --- [           main] o.s.batch.core.step.AbstractStep         : Step: [step] executed
in 83ms
2020-03-12 09:04:53.522  INFO 40522 --- [           main] o.s.b.c.l.support.SimpleJobLauncher      : Job: [SimpleJob:
[name=retryExceptionJob]] completed with the following parameters: [{}] and the following status: [COMPLETED] in 152ms
```

结果符合我们的预期。

假如通过`retryLimit(2)`将重试次数设置为2，并修改任务的名称为`retryExceptionJob1`，启动项目看看运行结果如何：

```
v2020-03-12 09:06:48.855  INFO 40610 --- [           main] o.s.b.c.l.support.SimpleJobLauncher      : Job: [SimpleJob:
[name=retryExceptionJob1]] launched with the following parameters: [{}]
2020-03-12 09:06:48.933  INFO 40610 --- [           main] o.s.batch.core.job.SimpleStepHandler     : Executing step: [step]
当前处理的数据：0
当前处理的数据：0
2020-03-12 09:06:48.979 ERROR 40610 --- [           main] o.s.batch.core.step.AbstractStep         : Encountered an error execu
step step in job retryExceptionJob1

org.springframework.retry.RetryException: Non-skippable exception in recoverer while processing; nested exception is
cc.mrbird.batch.exception.MyJobExecutionException: 任务处理出错
    at org.springframework.batch.core.step.item.FaultTolerantChunkProcessor$2.recover(FaultTolerantChunkProcessor.java:289) ~
[spring-batch-core-4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.retry.support.RetryTemplate.handleRetryExhausted(RetryTemplate.java:512) ~[spring-retry-
1.2.5.RELEASE.jar:na]
    at org.springframework.retry.support.RetryTemplate.doExecute(RetryTemplate.java:351) ~[spring-retry-1.2.5.RELEASE.jar:na]
    at org.springframework.retry.support.RetryTemplate.execute(RetryTemplate.java:211) ~[spring-retry-1.2.5.RELEASE.jar:na]
    at org.springframework.batch.core.step.item.BatchRetryTemplate.execute(BatchRetryTemplate.java:217) ~[spring-batch-core-
4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.batch.core.step.item.FaultTolerantChunkProcessor.transform(FaultTolerantChunkProcessor.java:298) ~
[spring-batch-core-4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.batch.core.step.item.SimpleChunkProcessor.process(SimpleChunkProcessor.java:210) ~[spring-batch-core
4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.batch.core.step.item.ChunkOrientedTasklet.execute(ChunkOrientedTasklet.java:77) ~[spring-batch-core-
4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.batch.core.step.tasklet.TaskletStep$ChunkTransactionCallback.doInTransaction(TaskletStep.java:407) ~
[spring-batch-core-4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.batch.core.step.tasklet.TaskletStep$ChunkTransactionCallback.doInTransaction(TaskletStep.java:331) ~
[spring-batch-core-4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.transaction.support.TransactionTemplate.execute(TransactionTemplate.java:140) ~[spring-tx-
5.2.4.RELEASE.jar:5.2.4.RELEASE]
    at org.springframework.batch.core.step.tasklet.TaskletStep$2.doInChunkContext(TaskletStep.java:273) ~[spring-batch-core-
4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.batch.core.scope.context.StepContextRepeatCallback.doInIteration(StepContextRepeatCallback.java:82)
[spring-batch-core-4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.batch.repeat.support.RepeatTemplate.getNextResult(RepeatTemplate.java:375) ~[spring-batch-infrastruc
4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.batch.repeat.support.RepeatTemplate.executeInternal(RepeatTemplate.java:215) ~[spring-batch-
infrastructure-4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.batch.repeat.support.RepeatTemplate.iterate(RepeatTemplate.java:145) ~[spring-batch-infrastructure-
4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.batch.core.step.tasklet.TaskletStep.doExecute(TaskletStep.java:258) ~[spring-batch-core-
4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.batch.core.step.AbstractStep.execute(AbstractStep.java:208) ~[spring-batch-core-
4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.batch.core.job.SimpleStepHandler.handleStep(SimpleStepHandler.java:148) [spring-batch-core-
4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.batch.core.job.AbstractJob.handleStep(AbstractJob.java:410) [spring-batch-core-
4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.batch.core.job.SimpleJob.doExecute(SimpleJob.java:136) [spring-batch-core-
4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.batch.core.job.AbstractJob.execute(AbstractJob.java:319) [spring-batch-core-
4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.batch.core.launch.support.SimpleJobLauncher$1.run(SimpleJobLauncher.java:147) [spring-batch-core-
4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.core.task.SyncTaskExecutor.execute(SyncTaskExecutor.java:50) [spring-core-
5.2.4.RELEASE.jar:5.2.4.RELEASE]
    at org.springframework.batch.core.launch.support.SimpleJobLauncher.run(SimpleJobLauncher.java:140) [spring-batch-core-
4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.8.0_231]
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62) ~[na:1.8.0_231]
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) ~[na:1.8.0_231]
    at java.lang.reflect.Method.invoke(Method.java:498) ~[na:1.8.0_231]
    at org.springframework.aop.support.AopUtils.invokeJoinpointUsingReflection(AopUtils.java:344) [spring-aop-
5.2.4.RELEASE.jar:5.2.4.RELEASE]
    at org.springframework.aop.framework.ReflectiveMethodInvocation.invokeJoinpoint(ReflectiveMethodInvocation.java:198) [sprin
aop-5.2.4.RELEASE.jar:5.2.4.RELEASE]
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:163) [spring-aop-
5.2.4.RELEASE.jar:5.2.4.RELEASE]
    at
org.springframework.batch.core.configuration.annotation.SimpleBatchConfiguration$PassthruAdvice.invoke(SimpleBatchConfiguration
:127) [spring-batch-core-4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:186) [spring-aop-
5.2.4.RELEASE.jar:5.2.4.RELEASE]
    at org.springframework.aop.framework.JdkDynamicAopProxy.invoke(JdkDynamicAopProxy.java:212) [spring-aop-
5.2.4.RELEASE.jar:5.2.4.RELEASE]
    at com.sun.proxy.$Proxy46.run(Unknown Source) [na:na]
    at org.springframework.boot.autoconfigure.batch.JobLauncherCommandLineRunner.execute(JobLauncherCommandLineRunner.java:192)
[spring-boot-autoconfigure-2.2.5.RELEASE.jar:2.2.5.RELEASE]
    at
org.springframework.boot.autoconfigure.batch.JobLauncherCommandLineRunner.executeLocalJobs(JobLauncherCommandLineRunner.java:16
[spring-boot-autoconfigure-2.2.5.RELEASE.jar:2.2.5.RELEASE]
    at
org.springframework.boot.autoconfigure.batch.JobLauncherCommandLineRunner.launchJobFromProperties(JobLauncherCommandLineRunner.
153) [spring-boot-autoconfigure-2.2.5.RELEASE.jar:2.2.5.RELEASE]
    at org.springframework.boot.autoconfigure.batch.JobLauncherCommandLineRunner.run(JobLauncherCommandLineRunner.java:148) [sp
boot-autoconfigure-2.2.5.RELEASE.jar:2.2.5.RELEASE]
    at org.springframework.boot.SpringApplication.callRunner(SpringApplication.java:784) [spring-boot-
2.2.5.RELEASE.jar:2.2.5.RELEASE]
    at org.springframework.boot.SpringApplication.callRunners(SpringApplication.java:768) [spring-boot-
2.2.5.RELEASE.jar:2.2.5.RELEASE]
    at org.springframework.boot.SpringApplication.run(SpringApplication.java:322) [spring-boot-2.2.5.RELEASE.jar:2.2.5.RELEASE]
    at org.springframework.boot.SpringApplication.run(SpringApplication.java:1226) [spring-boot-2.2.5.RELEASE.jar:2.2.5.RELEASE
    at org.springframework.boot.SpringApplication.run(SpringApplication.java:1215) [spring-boot-2.2.5.RELEASE.jar:2.2.5.RELEASE
    at cc.mrbird.batch.SpringBatchExceptionApplication.main(SpringBatchExceptionApplication.java:12) [classes/:na]
Caused by: cc.mrbird.batch.exception.MyJobExecutionException: 任务处理出错
    at cc.mrbird.batch.job.RetryExceptionJobDemo$1.process(RetryExceptionJobDemo.java:64) ~[classes/:na]
    at cc.mrbird.batch.job.RetryExceptionJobDemo$1.process(RetryExceptionJobDemo.java:55) ~[classes/:na]
    at org.springframework.batch.core.step.item.SimpleChunkProcessor.doProcess(SimpleChunkProcessor.java:134) ~[spring-batch-co
```

```
4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.batch.core.step.item.FaultTolerantChunkProcessor$1.doWithRetry(FaultTolerantChunkProcessor.java:233)
[spring-batch-core-4.2.1.RELEASE.jar:4.2.1.RELEASE]
    at org.springframework.retry.support.RetryTemplate.doExecute(RetryTemplate.java:287) ~[spring-retry-1.2.5.RELEASE.jar:na]
    ... 43 common frames omitted

2020-03-12 09:06:48.989  INFO 40610 --- [           main] o.s.batch.core.step.AbstractStep         : Step: [step] executed in 5
2020-03-12 09:06:49.019  INFO 40610 --- [           main] o.s.b.c.l.support.SimpleJobLauncher      : Job: [SimpleJob:
[name=retryExceptionJob1]] completed with the following parameters: [{}] and the following status: [FAILED] in 152ms
```

异常次数超过了重试次数，所以抛出了异常。

异常跳过

我们也可以在Step中配置异常跳过，即遇到指定类型异常时忽略跳过它，在job包下新建`SkipExceptionJobDemo`：

```
@Component
public class SkipExceptionJobDemo {

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

    @Bean
    public Job skipExceptionJob() {
        return jobBuilderFactory.get("skipExceptionJob")
                .start(step())
                .build();
    }

    private Step step() {
        return stepBuilderFactory.get("step")
                .<String, String>chunk(2)
                .reader(listItemReader())
                .processor(myProcessor())
                .writer(list -> list.forEach(System.out::println))
                .faultTolerant() // 配置错误容忍
                .skip(MyJobExecutionException.class) // 配置跳过的异常类型
                .skipLimit(1) // 最多跳过1次，1次过后还是异常的话，则任务会结束，
                // 异常的次数为reader，processor和writer中的总数，这里仅在processor里演示异
常跳过
                .build();
    }

    private ListItemReader<String> listItemReader() {
        ArrayList<String> datas = new ArrayList<>();
        IntStream.range(0, 5).forEach(i -> datas.add(String.valueOf(i)));
        return new ListItemReader<>(datas);
    }

    private ItemProcessor<String, String> myProcessor() {
        return item -> {
            System.out.println("当前处理的数据：" + item);
            if ("2".equals(item)) {
                throw new MyJobExecutionException("任务处理出错");
            } else {
                return item;
            }
        };
    }
}
```

在`step()`方法中，`faultTolerant()`表示开启容错功能，`skip(MyJobExecutionException.class)`表示遇到`MyJobExecutionException`异常时跳过，`skipLimit(1)`表示只跳过一次。

`myProcessor()`的逻辑是，当处理的item值为”2“的时候，抛出`MyJobExecutionException("任务处理出错")`异常。

此外我们还可以配置`SkipListener`类型监听器，在cc.mrbird.batch包下新建listener包，然后在该包下新建`MySkipListener`：

```
@Component
public class MySkipListener implements SkipListener<String, String> {
    @Override
    public void onSkipInRead(Throwable t) {
        System.out.println("在读取数据的时候遇到异常并跳过，异常：" + t.getMessage());
    }

    @Override
    public void onSkipInWrite(String item, Throwable t) {
        System.out.println("在输出数据的时候遇到异常并跳过，待输出数据：" + item + "，异常：" +
t.getMessage());
    }

    @Override
    public void onSkipInProcess(String item, Throwable t) {
        System.out.println("在处理数据的时候遇到异常并跳过，待输出数据：" + item + "，异常：" +
t.getMessage());
    }
}
```

然后将它注入到SkipExceptionJobDemo，并配置：

```
@Component
public class SkipExceptionJobDemo {

    ....

    @Autowired
    private MySkipListener mySkipListener;

    ....

    private Step step() {
        return stepBuilderFactory.get("step")
                .<String, String>chunk(2)
                .reader(listItemReader())
                .processor(myProcessor())
                .writer(list -> list.forEach(System.out::println))
                .faultTolerant() // 配置错误容忍
                .skip(MyJobExecutionException.class) // 配置跳过的异常类型
                .skipLimit(1) // 最多跳过1次，1次过后还是异常的话，则任务会结束，
                // 异常的次数为reader，processor和writer中的总数，这里仅在processor里演示异
常跳过
                .listener(mySkipListener)
                .build();
    }
    ....
}
```

启动项目，控制台日志打印如下：

```
2020-03-12 09:23:33.528  INFO 40759 --- [           main] o.s.b.c.l.support.SimpleJobLauncher      : Job: [SimpleJob:
[name=skipExceptionJob]] launched with the following parameters: [{}]
2020-03-12 09:23:33.664  INFO 40759 --- [           main] o.s.batch.core.job.SimpleStepHandler     : Executing step:
[step]
当前处理的数据：0
当前处理的数据：1
0
1
当前处理的数据：2
当前处理的数据：3
3
在处理数据的时候遇到异常并跳过，待输出数据：2，异常：任务处理出错
当前处理的数据：4
4
2020-03-12 09:23:33.854  INFO 40759 --- [           main] o.s.batch.core.step.AbstractStep         : Step: [step] executed
in 190ms
2020-03-12 09:23:33.885  INFO 40759 --- [           main] o.s.b.c.l.support.SimpleJobLauncher      : Job: [SimpleJob:
[name=skipExceptionJob]] completed with the following parameters: [{}] and the following status: [COMPLETED] in 324ms
```

事务问题

一次Setp分为Reader、Processor和Writer三个阶段，这些阶段统称为Item。默认情况下如果错误不是发生在Reader阶段，那么没必要再去重新读取一次数据。但是某些场景下需要Reader部分也需要重新执行，比如Reader是从一个JMS队列中消费消息，当发生回滚的时候消息也会在队列上重放，因此也要将Reader纳入到回滚的事物中，根据这个场景可以使用`readerIsTransactionalQueue()`来配置数据重读：

```
private Step step() {
    return stepBuilderFactory.get("step")
            .<String, String>chunk(2)
            .reader(listItemReader())
            .writer(list ->
list.forEach(System.out::println))
            .readerIsTransactionalQueue() // 消息队列数据重
读
            .build();
}
```

我们还可以在Step中手动配置事务属性，事物的属性包括隔离等级（isolation）、传播方式（propagation）以及过期时间（timeout）等：

```
private Step step() {
    DefaultTransactionAttribute attribute = new
DefaultTransactionAttribute();
    attribute.setPropagationBehavior(Propagation.REQUIRED.value());
    attribute.setIsolationLevel(Isolation.DEFAULT.value());
    attribute.setTimeout(30);

    return stepBuilderFactory.get("step")
            .<String, String>chunk(2)
            .reader(listItemReader())
            .writer(list -> list.forEach(System.out::println))
            .transactionAttribute(attribute)
            .build();
}
```

重启机制

默认情况下，任务执行完毕的状态为`COMPLETED`，再次启动项目，该任务的Step不会再执行，我们可以通过配置`allowStartIfComplete(true)`来实现每次项目重新启动都将执行这个Step：

```
private Step step() {
    return stepBuilderFactory.get("step")
            .<String, String>chunk(2)
            .reader(listItemReader())
            .writer(list ->
list.forEach(System.out::println))
            .allowStartIfComplete(true)
            .build();
}
```

某些Step可能用于处理一些先决的任务，所以当Job再次重启时这Step就没必要再执行，可以通过设置`startLimit()`来限定某个Step重启的次数。当设置为1时候表示仅仅运行一次，而出现重启时将不再执行：

```
private Step step() {
    return stepBuilderFactory.get("step")
            .<String, String>chunk(2)
            .reader(listItemReader())
            .writer(list ->
list.forEach(System.out::println))
            .startLimit(1)
            .build();
}
```

部分内容参考自：https://blog.csdn.net/sswltt/article/details/103817645

> 本章源码链接：https://github.com/wuyouzhuguli/SpringAll/tree/master/72.spring-batch-exception。