使用springbatch实现数据迁移实例

Table of Contents

数据迁移Data migration概述

在实际的软件产品开发过程当中，由于软件的不停迭代和升级，或者是一些商业上，战略上的策略调整，我们的系统总是有可能会遇到需要从一个数据库产品迁移到另一个数据库上的场景。这意味着我们不得不将以前的数据迁移到新的数据库上去，因为我们不可能因为搬迁平台而丢掉用户的数据，这对于任何一个产品来说都是一件不可能被允许的事情。

数据迁移的类型及其挑战

数据迁移过程分三个级别执行。

Storage migration 存储迁移

通过技术更新证明存储迁移是合理的，并且该过程被用作通过识别过时或损坏数据来进行数据验证和减少的最佳时间。 该过程涉及将存储和文件块从一个存储系统移动到另一个存储系统，无论是在磁盘，磁带还是云上。 有许多存储迁移产品和工具可以帮助我们顺利完成整个过程。 存储迁移还提供了修复任何孤立存储或低效的机会。

Database migration 数据库迁移

当需要更改数据库供应商，升级数据库软件或将数据库移动到云时，可以完成数据库迁移。 在这种类型的迁移中，底层数据可能会发生变化，这会在协议或数据语言发生变化时影响应用程序层。 数据库中的数据迁移涉及修改数据而不更改模式。 一些关键任务包括评估数据库大小以确定需要多少存储，测试应用程序以及保证数据机密性。 迁移过程中可能会出现兼容性问题，因此首先测试该过程非常重要。

Application migration 应用迁移

切换到其他供应商应用程序或平台时，可能会发生应用程 这个过程有其固有的复杂层，因为应用程序与其他应用程序交互，每个应用程序都有自己的数据模型。 应用程序不是为便携式而设计的。 管理工具，操作系统和虚拟机配置都可以与开发或部署应用程序的环境不同。 成功的应用程序迁移可能需要使用中间件产品来弥合技术差距。

云迁移是一项主要的技术趋势，因为云为内部部署基础架构提供按需灵活性，可扩展性和Capex的减少。 公共云提供商为存储，数据库和应用程序迁移提供各种服务。

一个数据迁移代码实例

下面这个例子基于spring boot和maven，实现的功能是从file当中读取数据，同时写入到另一个文件里面。在实际做数据迁移的时候，只需要将代码当中读取数据，写数据的相应逻辑替换即可，流程和框架是一样的。

maven依赖如下:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>io.spring.cloud.dataflow.ingest</groupId>
    <artifactId>ingest</artifactId>
    <version>1.0.0.BUILD-SNAPSHOT</version>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.0.RELEASE</version>
    </parent>
    <properties>
        <java.version>1.8</java.version>
        <maven.compiler.plugin.version>3.7.0</maven.compiler.plugin.version>
        <spring.cloud.task.version>1.2.2.RELEASE</spring.cloud.task.version>
        <checkstyle.config.location>checkstyle.xml</checkstyle.config.location>
        <checkstyle.plugin.version>2.17</checkstyle.plugin.version>
    </properties>
```

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-batch</artifactId>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-task-core</artifactId>
        <version>${spring.cloud.task.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-task-batch</artifactId>
        <version>${spring.cloud.task.version}</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>${maven.compiler.plugin.version}</version>
            <configuration>
                <source>${java.version}</source>
                <target>${java.version}</target>
                <testSource>${java.version}</testSource>
                <testTarget>${java.version}</testTarget>
                <compilerArgument>-Xlint:all</compilerArgument>
            </configuration>
        </plugin>
    </plugins>
</build>
<repositories>
    <repository>
        <id>repository.spring.milestone</id>
        <name>Spring Milestone Repository</name>
        <url>http://repo.spring.io/milestone</url>
    </repository>
</repositories>
<pluginRepositories>
    <pluginRepository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone</url>
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
```

```
        </pluginRepository>
    </pluginRepositories>
    <reporting>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-checkstyle-plugin</artifactId>
                <version>${checkstyle.plugin.version}</version>
            </plugin>
        </plugins>
    </reporting>
</project>
```

启动类Application如下：

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

在启动类的注解当中我们加上了注解@EnableTask，这个注解可以让我们使用spring cloud data flow的功能。关于spring cloud data flow 的功能可以参考后面的部分内容。

Batch批处理的核心逻辑定义如下：

```
/*
 * Copyright 2018 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *        http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package io.spring.cloud.dataflow.ingest.config;

import io.spring.cloud.dataflow.ingest.domain.Person;
import io.spring.cloud.dataflow.ingest.listner.JobCompletionNotificationListener;
import io.spring.cloud.dataflow.ingest.mapper.fieldset.PersonFieldSetMapper;
import io.spring.cloud.dataflow.ingest.processor.PersonItemProcessor;
import javax.sql.DataSource;

import io.spring.cloud.dataflow.ingest.processor.PersonItemProcessor2;
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.launch.support.RunIdIncrementer;
import org.springframework.batch.item.ItemProcessor;
import org.springframework.batch.item.ItemStreamReader;
```

```
import org.springframework.batch.item.ItemWriter;
import org.springframework.batch.item.database.JdbcBatchItemWriter;
import org.springframework.batch.item.database.builder.JdbcBatchItemWriterBuilder;
import org.springframework.batch.item.file.FlatFileItemWriter;
import org.springframework.batch.item.file.builder.FlatFileItemReaderBuilder;

import org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor;
import org.springframework.batch.item.file.transform.DelimitedLineAggregator;
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.ClassPathResource;
import org.springframework.core.io.FileSystemResource;
import org.springframework.core.io.ResourceLoader;

/**
 * Class used to configure the batch job related beans.
 *
 * @author Chris Schaefer
 */
@Configuration
@EnableBatchProcessing
public class BatchConfiguration {
 private final DataSource dataSource;
 private final ResourceLoader resourceLoader;
 private final JobBuilderFactory jobBuilderFactory;
 private final StepBuilderFactory stepBuilderFactory;

 @Autowired
 public BatchConfiguration(final DataSource dataSource, final JobBuilderFactory jobBuilderFactory,
   final StepBuilderFactory stepBuilderFactory,
   final ResourceLoader resourceLoader) {
  this.dataSource = dataSource;
  this.resourceLoader = resourceLoader;
  this.jobBuilderFactory = jobBuilderFactory;
  this.stepBuilderFactory = stepBuilderFactory;
 }


 @Bean
 @StepScope
 public ItemStreamReader<Person> reader() throws Exception {
  return new FlatFileItemReaderBuilder<Person>()
    .name("reader")
    .resource(new ClassPathResource("data.csv"))
    .delimited()
    .names(new String[] {"firstName", "lastName"})
    .fieldSetMapper(new PersonFieldSetMapper())
    .build();
 }

 @Bean
 public ItemProcessor<Person, Person> processor() {
  return new PersonItemProcessor();
 }

 @Bean
 public ItemProcessor<Person, Person> processor2() {
  return new PersonItemProcessor2();
 }

 @Bean
 public ItemWriter<Person> writer() {
  return new JdbcBatchItemWriterBuilder<Person>()
```

```
  .beanMapped()
  .dataSource(this.dataSource)
  .sql("INSERT INTO people (first_name, last_name) VALUES (:firstName, :lastName)")
  .build();
}


@Bean
public ItemWriter<Person> writerToFile() {
 //Create writer instance
 FlatFileItemWriter<Person> writer = new FlatFileItemWriter<>();

 //Set output file location
 writer.setResource(new FileSystemResource("output/outputData.csv"));

 //All job repetitions should "append" to same output file
 writer.setAppendAllowed(true);

 //Name field values sequence based on object properties
 writer.setLineAggregator(new DelimitedLineAggregator<Person>() {
  {
   setDelimiter(",");
   setFieldExtractor(new BeanWrapperFieldExtractor<Person>() {
    {
     setNames(new String[] { "firstName", "lastName" });
    }
   });
  }
 });
 return writer;
}

@Bean
public Job ingestJob(JobCompletionNotificationListener listener) throws Exception {
 return jobBuilderFactory.get("ingestJob")
  .incrementer(new RunIdIncrementer()).listener(listener)
  .flow(step1()).next(step2())
  .end()
  .build();
}

@Bean
public Step step1() throws Exception {
 return stepBuilderFactory.get("ingest")
  .<Person, Person>chunk(10)
  .reader(reader())
  .processor(processor())
  //.writer(writer())
  .writer(writerToFile())
  .build();
}

@Bean
public Step step2() throws Exception {
 return stepBuilderFactory.get("step2")
   .<Person, Person>chunk(10)
   .reader(reader())//get the data from anywhere in the reader bean
   .processor(processor2())//do the process for the data with the logic defined in processor bean process2
   //.writer(writer())//write the processed data into the as the writer bean defined,
   .writer(writerToFile())
   .build();
 }
}
```

上面的代码是spring batch的核心实现，定义了batch的job和step，以及数据源等信息，实现了写入到文件，或者是写入数据库的功能，同时定义了两个迁移过程的两个step，两个step将按照我们指定的顺序执行。每一个step的reader，processor，和writer都可以是自己独有的逻辑。如果我们想把数据写入到数据库或者是从数据库读数据，则只需要定义好对应的DataSource信息，然后再reader和writer里指定DataSource和sql语句即可。

model类的代码如下：

```
public class Person {
 private final String firstName;
 private final String lastName;

 public Person(final String firstName, final String lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
 }

 public String getFirstName() {
  return firstName;
 }

 public String getLastName() {
  return lastName;
 }

 @Override
 public String toString() {
  return "First name: " + firstName + " , last name: " + lastName;
 }
}
```

Mapper类如下

```
import io.spring.cloud.dataflow.ingest.domain.Person;

import org.springframework.batch.item.file.mapping.FieldSetMapper;
import org.springframework.batch.item.file.transform.FieldSet;

/**
 * Maps the provided FieldSet into a Person object.
 *
 */
public class PersonFieldSetMapper implements FieldSetMapper<Person> {
 @Override
 public Person mapFieldSet(FieldSet fieldSet) {
  String firstName = fieldSet.readString(0);
  String lastName = fieldSet.readString(1);

  return new Person(firstName, lastName);
 }
}
```

Mapper类的功能主要是作为一个数据的载体，我们使用java技术将数据从数据源读取到之后，数据是存放到结果集ResultSet当中的，需要有一个地方去接收它，并把这条数据转化成我们需要的格式或者是方便我们处理的格式等。

两个step的简单处理逻辑分别定义如下：

```
public class PersonItemProcessor implements ItemProcessor<Person, Person> {
 private static final Logger LOGGER = LoggerFactory.getLogger(PersonItemProcessor.class);

 @Override
 public Person process(Person person) throws Exception {
  String firstName = person.getFirstName().toUpperCase();
  String lastName = person.getLastName().toUpperCase();

  Person processedPerson = new Person(firstName, lastName);

  LOGGER.info("Processed: " + person + " into: " + processedPerson);

  return processedPerson;
 }
}
```

```
import io.spring.cloud.dataflow.ingest.domain.Person;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.item.ItemProcessor;

public class PersonItemProcessor2 implements ItemProcessor<Person, Person> {

    private static final Logger log = LoggerFactory.getLogger(PersonItemProcessor.class);

    @Override
    public Person process(final Person person) throws Exception {
        final String firstName = (person.getFirstName()+" step2").toUpperCase();
        final String lastName = (person.getLastName()+" step2").toUpperCase();

        final Person transformedPerson = new Person(firstName, lastName);

        log.info("Converting (" + person + ") into (" + transformedPerson + ")");

        //test the exception case
       /* if(firstName.equals("JANE STEP2")){
            throw new RuntimeException("I am a exception");
        }*/
        return transformedPerson;
    }

}
```

整个job的listener定义如下：

```
import io.spring.cloud.dataflow.ingest.domain.Person;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.batch.core.BatchStatus;
import org.springframework.batch.core.JobExecution;
import org.springframework.batch.core.listener.JobExecutionListenerSupport;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component
public class JobCompletionNotificationListener extends JobExecutionListenerSupport {

    private static final Logger log = LoggerFactory.getLogger(JobCompletionNotificationListener.class);

    private final JdbcTemplate jdbcTemplate;

    @Autowired
    public JobCompletionNotificationListener(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Override
    public void afterJob(JobExecution jobExecution) {
        if(jobExecution.getStatus() == BatchStatus.COMPLETED) {
            log.info("!!! JOB FINISHED! Time to verify the results");

            jdbcTemplate.query("SELECT first_name, last_name FROM people",
                    (rs, row) -> new Person(
                            rs.getString(1),
                            rs.getString(2))
            ).forEach(person -> log.info("Found <" + person + "> in the database."));
        }
    }
}
```

整个job的listener的功能是在job执行完成之后去执行的一段逻辑，在整个job执行完成之后，我们也许需要执行迁移数据验证等功能，就可以放在这个地方做，除了job执行完之后的listener之外，spring batch还提供了非常丰富的listener来满足我们的需求，比如在read数据的前后，write数据的前后，process数据的前后，我们都是可以定义listener去做我们需要的事情的。

spring cloud data flow介绍

spring cloud data flow也是spring的一个子项目，用于管理各种数据流，它还提供了一个图形化界面供我们使用，spring batch的job也可以注册在spring cloud data flow上。在本地的安装使用流程如下。

使用wget命令下载两个jar包

```
wget http://repo.spring.io/release/org/springframework/cloud/spring-cloud-dataflow-server-local/1.7.3.RELEASE/spring-cloud-dataflow-server-local-1.7.3
```

```
wget http://repo.spring.io/release/org/springframework/cloud/spring-cloud-dataflow-shell/1.7.3.RELEASE/spring-cloud-dataflow-shell-1.7.3.RELEASE.ja
```

在下载好之后，进入jar包所在目录，直接使用命令启动。

先启动server

```
java -jar spring-cloud-dataflow-server-local-1.7.3.RELEASE.jar
```

如果不想使用它的默认内存数据库，可以在bash窗口中通过如下方式指定:

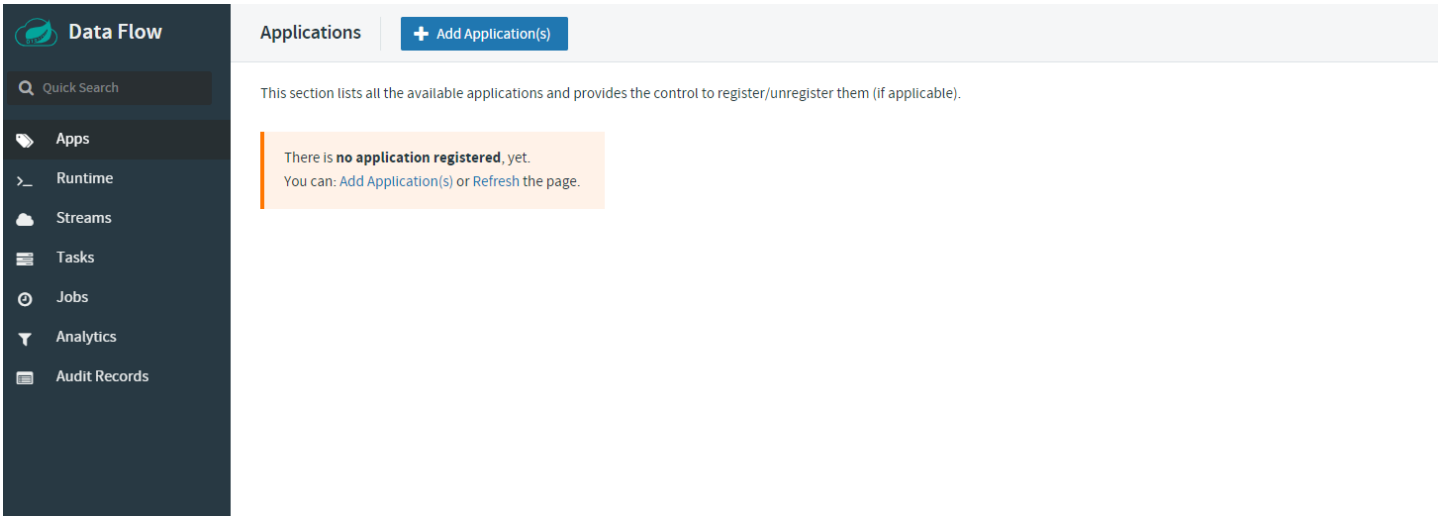
```
java -jar spring-cloud-dataflow-server-local-1.7.3.RELEASE.jar \
  --spring.datasource.url=jdbc:postgresql://localhost:5432/DBName\
  --spring.datasource.username=XXX\
  --spring.datasource.password=XXX \
  --spring.datasource.driver-class-name=org.postgresql.Driver
```

再启动shell

```
java -jar spring-cloud-dataflow-shell-1.7.3.RELEASE.jar
```

启动完成之后，如果shell的窗口显示server unknown，则我们可以通过以下命令指定它的server端口:

```
dataflow config server+(server地址)
```



由于当前我们没有注册任何app和jobs，所以这里什么也没有，我们可以直接在dashboard上添加app，也可以在命令行在task shell里通过如下命令注册app:

```
app register --type source --name my-app --uri file://root/apps/my-app.jar
```

在spring cloud data flow里我们可以管理我们的APP，控制他们的启动，检测job的运行状况等，这些数据会被保存在数据库里，

默认情况下，spring cloud data flow server会使用一个h2内存数据库存储我们定义的job等任务，我们也可以配置使用其他数据库。关于spring cloud data flow的更多内容，这里就不做过多的介绍，可以参考spring的。遇到问题可以自己google解决。

在使用Spring Bacth以及Spring Cloud data flow时遇到的问题收集

问题1

启动spring batch项目报错: org.postgresql.util.PSQLException: ERROR: relation "batch_job_instance" does not exist?

答案: 在使用spring boot时则在application.properties中加上: spring.batch.initialize-schema=ALWAYS

问题2

启动spring batch项目时batch相关的configuration报错: org.springframework.batch.item.ReaderNotOpenException: Reader must be open before it can be read.

把方法的返回值类型改为ItemReader解决了这个问题，参考上的回复。

问题3

error信息: NoSuchMethodError: org.springframework.boot.builder.SpringApplicationBuilder.<init>([Ljava/lang/Object;)V
原因：spring boot和spring cloud版本不兼容.
解决办法：遵守spring boot和spring cloud的版本兼容，下表:

Spring Cloud	Spring Boot
Finchley	兼容Spring Boot 2.0.x，不兼容Spring Boot 1.5.x
Dalston和Edgware	兼容Spring Boot 1.5.x，不兼容Spring Boot 2.0.x
Camden	兼容Spring Boot 1.4.x，也兼容Spring Boot 1.5.x
Brixton	兼容Spring Boot 1.3.x，也兼容Spring Boot 1.4.x
Angel	兼容Spring Boot 1.2.x

问题4

在windows的cmd里面执行启动sping cloud data flow jar ， 报错：Caused by: java.lang.RuntimeException: Driver org.h2.Driver claims to not accept jdbcUrl, jdbc:postgresql://localhost:5432/test。
但是在bash里面成功,
原因：命令是bash格式的java命令

问题5

在spring batch里面定义了两个datasource的bean之后，启动error: java.lang.IllegalStateException: To use the default TaskConfigurer the context must contain no more than one DataSource, found 2。

解决办法：spring batch官方提供了解决办法，在有一个以上的datasource的时候，我们需要配置一个CustomTaskConfigurer ， 参考spring的

```
import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.cloud.task.configuration.DefaultTaskConfigurer;
import org.springframework.stereotype.Component;

@Component
public class CustomTaskConfigurer extends DefaultTaskConfigurer {

 @Autowired
 public CustomTaskConfigurer(@Qualifier("secondDataSource") DataSource dataSource) {
  super(dataSource);
 }
}
```

问题6

在定义spring的data souce bean之后报错:Caused by: java.lang.IllegalArgumentException: dataSource or dataSourceClassName or jdbcUrl is required。

其中data source的bean定义如下:

```
@Bean
   @ConfigurationProperties("spring.datasource")
   public DataSource sourceDataSource() {
      return DataSourceBuilder.create().build();
   }
```

原因: 在类路径上有Hikari，因为Hikari没有url属性(但是确实有一个jdbcUrl属性)。在这种情况下，必须重写配置如下:

```
spring.datasource.jdbc-url=jdbc:postgresql://localhost/dbname
```

问题7

在spring batch里面定义了两个datasource的bean之后，启动error: java.lang.IllegalStateException:
Parameter 0 of method setDataSource in org.springframework.batch.core.configuration.annotation.DefaultBatchConfigurer required a single bean, but 2 were found。

解决办法，指定一个bean为primary，即加上@Primary注解

问题8

如何让spring在启动的时候不执行batch job

Spring also automatically run batch jobs configured. To disable auto-run of jobs, you need to use spring.batch.job.enabled property in application.properties file.

application.properties:

```
spring.batch.job.enabled=false
```

问题9

使用spring batch的 jobRegistry.getJob(jobName)方法报错:NoSuchJobException: No job configuration with the name.
解决办法: 将jobRegistry使用spring的@Autowired的方式注入进来即可

参考