软件体系结构作业

课程名称： 软件体系结构

班 级： 软工2005班

学 号： 2205059039

姓 名： 杨胜霜

日 期： 2023- 4- 23

目录

[第一次作业 日志封装 6](#_Toc24403)

[1.1 需求场景 6](#_Toc5584)

[1.2 解答方法 6](#_Toc23082)

[1.3 解答 6](#_Toc8220)

[1.3.1 新建maven工程，在pom文件中添加log4j2的依赖 6](#_Toc30196)

[1.3.2 将日志文件写到控制台 9](#_Toc6992)

[1.3.3 将日志文件写进指定文件。 9](#_Toc7361)

[1.3.4 测试结果 13](#_Toc20817)

[第二次作业 异常划分 15](#_Toc8362)

[1.4 需求场景 15](#_Toc25716)

[1.5 解答方案 15](#_Toc22730)

[1.6 解答 15](#_Toc32023)

[1.6.1 服务异常 16](#_Toc2801)

[1.6.2 应用异常 16](#_Toc13587)

[第三次作业 Spring多线程 17](#_Toc12238)

[1.7 需求场景 17](#_Toc14246)

[1.8 解答方案 17](#_Toc19889)

[1.9 解答 17](#_Toc455)

[1.9.1 创建一个配置类 17](#_Toc17937)

[1.9.2 编写线程执行类 18](#_Toc19970)

[1.9.3 测试类 19](#_Toc5233)

[1.9.4 运行结果 19](#_Toc20558)

[第四次作业 springboot集成 19](#_Toc12647)

[1.10 需求场景 19](#_Toc18813)

[1.11 解答方案 20](#_Toc8904)

[1.12 解答 20](#_Toc30512)

[1.12.1 新建项目 20](#_Toc19017)

[1.12.2 配置yml,mqconfig 20](#_Toc19581)

[第五次作业 流程引擎 22](#_Toc19989)

[1.13 需求场景 22](#_Toc162)

[1.14 解答方案 23](#_Toc4904)

[1.15 解答 24](#_Toc11391)

[1.15.1 下载，导入项目。 24](#_Toc16240)

[1.15.2 修改配置文件。 24](#_Toc13483)

[1.15.3 找到资源文件下的这三个文件。 24](#_Toc10753)

[1.15.4 开发环境配置。 24](#_Toc27996)

[1.15.5 启动本机mysql服务 25](#_Toc4577)

[1.15.6 找到ui包里的pagejson文件，启动前台。 25](#_Toc3717)

[1.15.7 控制台可以看到已启动服务。 25](#_Toc30770)

[1.15.8 启动项目。 26](#_Toc459)

[1.15.9 登录若依 26](#_Toc8399)

[第六次作业 JDK Timer调度 26](#_Toc8518)

[1.16 需求场景 26](#_Toc5260)

[1.17 解答方案 26](#_Toc29992)

[1.18 解答 27](#_Toc6843)

[1.18.1 TaskQueue：任务队列 27](#_Toc12224)

[1.18.2 TimerTask：任务实体 27](#_Toc2985)

[1.18.3 TimerThread：事件消费者 28](#_Toc25204)

[1.18.4 测试代码 28](#_Toc11310)

[1.18.5 测试结果 29](#_Toc31653)

[第七次作业 Spring IOC容器 29](#_Toc640)

[1.19 需求场景 29](#_Toc23708)

[1.20 解答方案 31](#_Toc22255)

[1.21 解答 32](#_Toc4079)

[1.21.1 新建项目。 32](#_Toc26422)

[1.21.2 修改pom文件。 32](#_Toc11262)

[1.21.3 导入代码。 33](#_Toc18810)

[1.21.4 测试 33](#_Toc2686)

[第八次作业 MVC架构 34](#_Toc9272)

[1.22 需求场景 34](#_Toc1321)

[1.23 解答方案 35](#_Toc19118)

[1.24 解答 35](#_Toc10499)

[1.24.1 新建MVC模块 35](#_Toc2438)

[1.24.2 添加依赖项 35](#_Toc7396)

[1.24.3 新建包名 36](#_Toc3526)

[1.24.4 添加代码 36](#_Toc7454)

[1.24.5 测试结果 38](#_Toc24658)

[第九次作业 SpringAOP 38](#_Toc15731)

[1.25 需求场景 38](#_Toc650)

[1.26 解答方案 40](#_Toc27587)

[1.27 解答 40](#_Toc23753)

[1.27.1 创建aop包 40](#_Toc16661)

[1.27.2 在aop包下创建，注解包lang,代理包proxy 40](#_Toc30150)

[1.27.3 创建切面代理抽象类 40](#_Toc23522)

[1.27.4 创建代理接口 44](#_Toc25137)

[1.27.5 创建代理链 44](#_Toc5585)

[1.27.6 创建代理管理器 46](#_Toc5628)

[1.27.7 创建事务代理 48](#_Toc14407)

[1.27.8 编写lang注解类。 51](#_Toc19764)

# 第一次作业 日志封装

## 需求场景

1、通过日志文件记录系统运行信息。

2、封装日志，提供对常见日志系统的扩展。

## 解答方法

1、引入slf4j和Lombok包，具体日志组件留给用户自己扩展。框架设计如下：

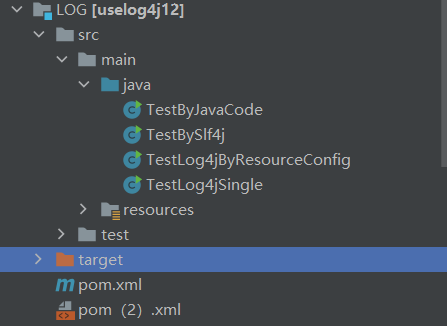


2、引入切面，进行日志记录。

## 解答

### 新建maven工程，在pom文件中添加log4j2的依赖

1）新建工程结构如下



2.添加依赖

<properties>

<maven.compiler.source>8</maven.compiler.source>

<maven.compiler.target>8</maven.compiler.target>

<log4j.version>2.5</log4j.version>

</properties>

<dependencies>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.10</version>

</dependency>

<dependency> <!-- 桥接：告诉Slf4j使用Log4j2 -->

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-slf4j-impl</artifactId>

<version>2.2</version>

</dependency>

<dependency> <!-- 桥接：告诉commons logging使用Log4j-->

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-jcl</artifactId>

<version>2.2</version>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-api</artifactId>

<version>${log4j.version}</version>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-core</artifactId>

<version>${log4j.version}</version>

</dependency>

<!-- log4j end-->

</dependencies>

### 将日志文件写到控制台

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class TestBySlf4j {

private static final Logger logger= LoggerFactory.getLogger(TestBySlf4j.class);

public static void main(String[] args) {

logger.info("infoslf");

logger.debug("debugslf");

logger.error("errorslf");

logger.warn("warnslf");

}

}

### 将日志文件写进指定文件。

1）编写log4j2.xml配置文件。

<?xml version="1.0" encoding="UTF-8"?>

<Configuration status="WARN">

<properties>

<property name="LOG\_HOME">D:/86158/logs</property>//指定日志存放目录

<property name="FILE\_NAME">mylog</property>

</properties>

<Appenders>

<Console name="Console" target="SYSTEM\_OUT">

<PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n" />

</Console>

<File name="FileAppender" fileName="D:/86158/logs/app.log">

//指定日志在目标目录生成指定日志文件

<PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n" />

</File>

<RollingRandomAccessFile name="RollingRandomAccessFile" fileName="D:/86158/logs/apprandom.log"

//指定日志在目标目录生成指定日志文件

filePattern="${LOG\_HOME}/$${date:yyyy-MM}/${FILE\_NAME}-%d{yyyy-MM-dd HH-mm}-%i.log">

<PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>

<Policies>

<TimeBasedTriggeringPolicy interval="1"/>

<SizeBasedTriggeringPolicy size="10 MB"/>

</Policies>

<DefaultRolloverStrategy max="20"/>

</RollingRandomAccessFile>

<!-- 发现Async 好像PatternLayout的输出格式配置的和输出的格式不一样，不用异步就完全一样 -->

<Async name="AsyncAppender">

<AppenderRef ref="FileAppender"/>

</Async>

</Appenders>

<Loggers>

<Root level="info">

<AppenderRef ref="Console" />

<AppenderRef ref="AsyncAppender"/>

<AppenderRef ref="RollingRandomAccessFile"/>

</Root>

</Loggers>

</Configuration>

2）代码编写（通过代码指定文件位置）

import org.apache.logging.log4j.core.config.Configurator;

import java.io.\*;

public class TestByJavaCode {

public static void main(String[] args) throws IOException {

File file = new File("D:/log4j2.xml");

BufferedInputStream in = new BufferedInputStream(new FileInputStream(file));

final ConfigurationSource source = new ConfigurationSource(in);

Configurator.initialize(null, source);

Logger logger = LogManager.getLogger("myLogger");

logger.info("info11");

logger.debug("debug11");

logger.error("error11");

logger.warn("warn11");

logger.fatal("fatal11");

}

}

3）通过配置文件指定文件的位置

import org.apache.logging.log4j.LogManager;

import org.apache.logging.log4j.Logger;

public class TestLog4jByResourceConfig {

public static void main(String[] args) {

Logger logger= LogManager.*getLogger*();

logger.info("info");

logger.debug("debug");

logger.error("error");

logger.warn("warn");

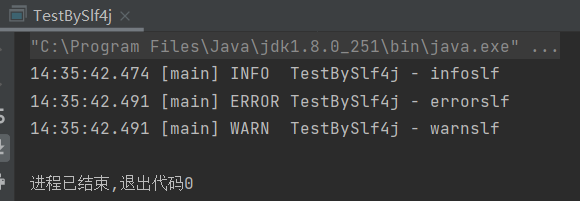
logger.fatal("fatal");

}

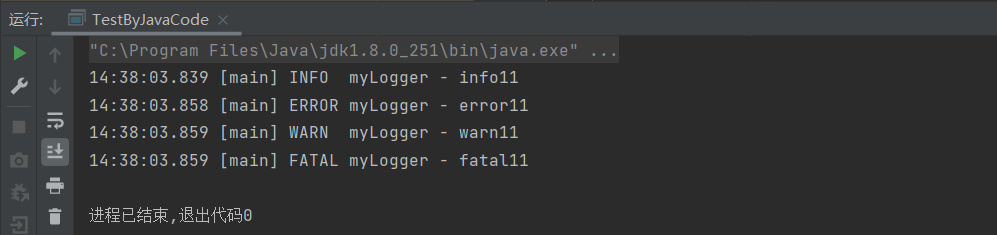
}

### 测试结果

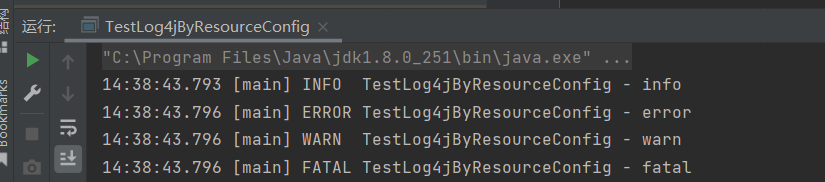
1）TestBySlf4j



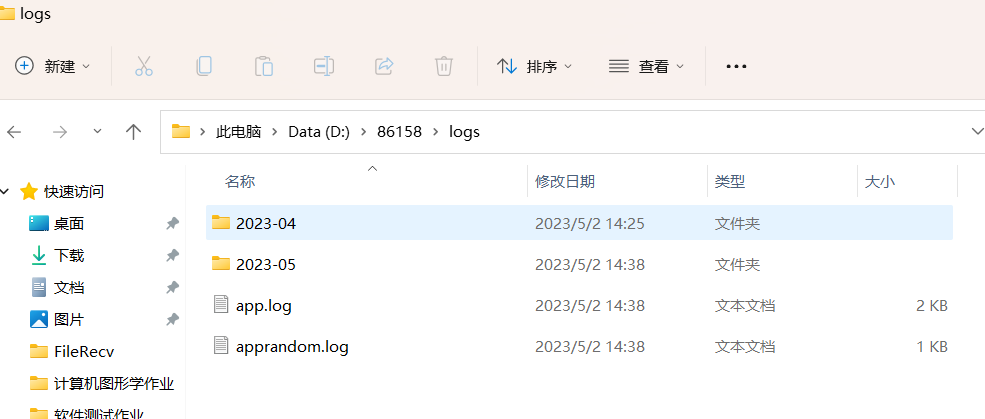
2）TestByJavaCode



3）TestLog4jByResourceConfig



4）生成日志如下



# 第二次作业 异常划分

## 需求场景

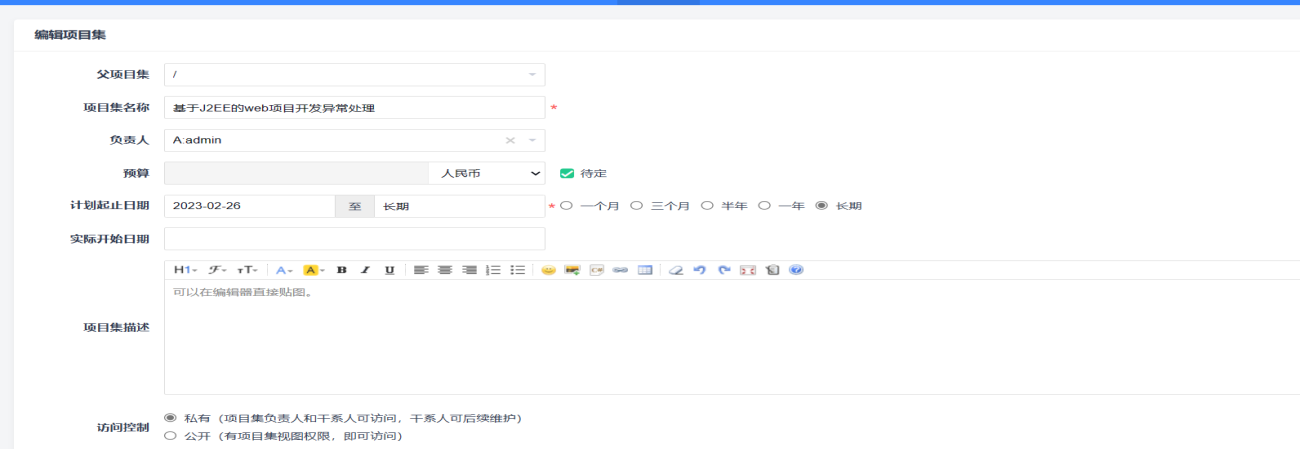
将平台异常划分为服务异常和应用异常，并且各个模块异常定义异常编号，搭建平台的基础异常。

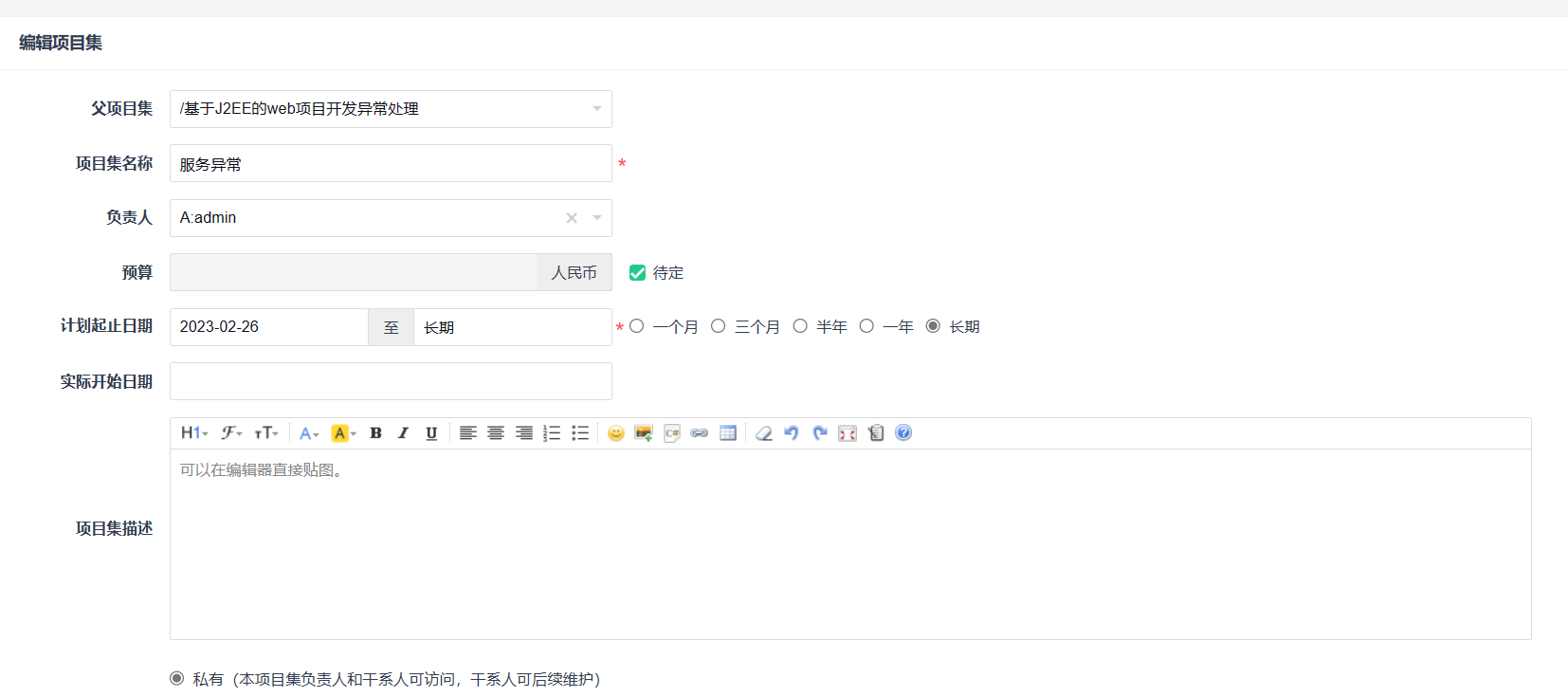
## 解答方案

1. 进入禅道。
2. 创建一个项目集名为平台异常
3. 再在项目集下面分别创建我们的服务异常和应用异常
4. 将出现的异常分别放在我们分类好的异常下面

## 解答

服务异常。创建项目集名为基于J2EE的web项目异常，并在其下面创建一个名为服务异常的





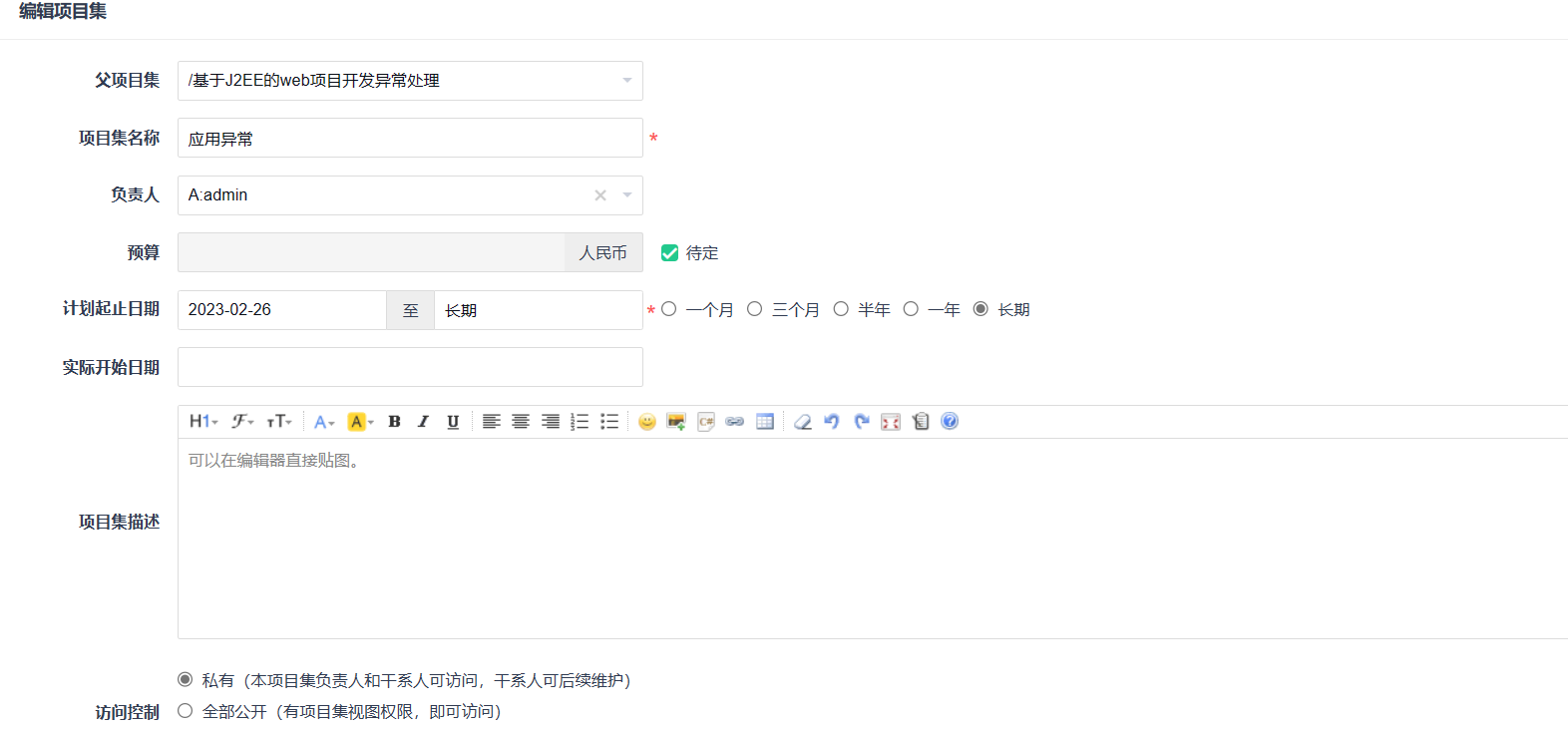
### 服务异常

包括IP地址服务，Tomcat服务，SQL服务等等，分别创建归属于服务异常当中。



### 应用异常

前面服务异常操作相同，而应用异常包括，类不存在，语法不正确，语句错误等等。





# 第三次作业 Spring多线程

## 需求场景

连接封装的对象中的多线程技术

## 解答方案

1. 了解线程

https://blog.csdn.net/H215919719/article/details/126566319

1. 了解线程框架

https://blog.csdn.net/qq\_42719291/article/details/121734165

1. 线程池的创建
2. 线程管理

## 解答

Spring 通过任务执行器来实现多线程和并发编程。实际上开发中的任务一般是非阻塞的，即异步的，所以我们要在配置类中通过@EnableAsync注解来开启对异步任务的支持，并通过在实际执行的Bean的方法上使用@Async注解来声明一个异步任务。

### 创建一个配置类

@Override

public Executor getAsyncExecutor(){

ThreadPoolTaskExecutor taskExecutor = new ThreadPoolTaskExecutor

//核心线程数

taskExecutor.setCorePoolS1ze(5)

//最大线程数

taskExecutor.setMaxPoolSize(50);

//队列最大长度

taskExecutor.setQueueCapacity(1000);

//线程池维护线程所允许的空间时间

taskExecutor.setKeepALiveSeconds(120);

//线程池对拒绝任务的处理策略

taskExecutor.setRejectedExecutionHandler(new ThreadPoolExecutor.AbortPolicy());

taskExecutor.initialize0)

return taskExecutor;

@Bean

public Executor getThreadPool() {

ThreadPoolTaskExecutor taskExecutor = new ThreadPoolTaskExecutortaskExecutor.setCorePoolSize(5);

taskExecutor.setMaxPoolS1ze(1000) ;

taskExecutor.setQueueCapacity(1000);

taskExecutor.initialize();

return taskExecutor;

### 编写线程执行类

public class DemoThreadService

QResource(name = “getThreadPool”)

private Executor executor;

@Async

public vo1d executeAsyncTask(Integer i) {

System.out.println("执行异步任务：" + 1);

}

public void executeAsyncTaskPlu(Integer i) {

executor.execute(new Runnable(){

@verride

public void run() {

System.out.println("执行异步任务+1" + i);

}

})；

}

}

### 测试类

publ1c class SpringThreadTest {

AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(ThreadConfig.cLass);

@Test

public vo1d contextTest(){

DenoThreadService denoThreadService = context.getBean(DemoThreadService .cLass);

for(int i = 0; i<1000; i++){

denoThreadService.executeAsyncTaskPlus

denoThreadService.executeAsyncTask();

@After

public void closeContext()

{

context.close();

}

}

### 运行结果



显然同时执行了多个任务，成功创建多线程

# 第四次作业 springboot集成

## 需求场景

主要原因是由于在高并发环境下，由于来不及同步处理，请求往往会发生堵塞，比如说，大量的insert，update之类的请求同时到达MySQL，直接导致无数的行锁表锁，甚至最后请求会堆积过多，从而触发too many connections错误。通过使用消息队列，我们可以异步处理请求，从而缓解系统的压力。

## 解答方案

1、下载activemq

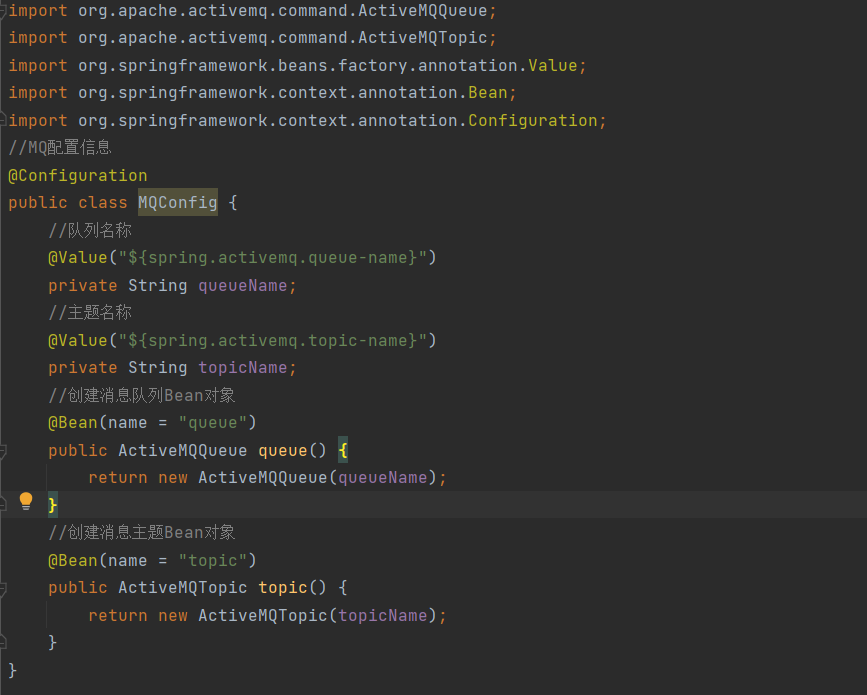
2、导入idea

3、启动

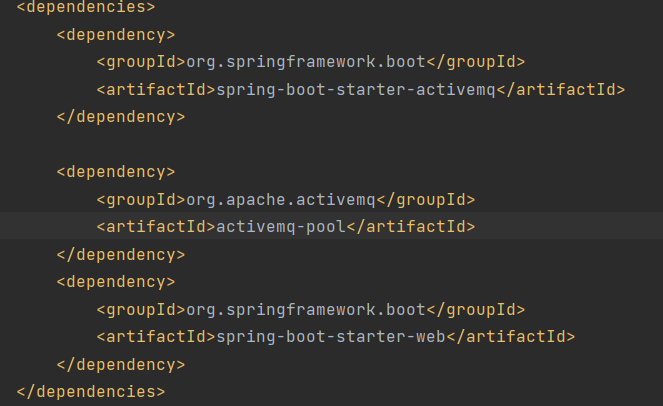
## 解答

### 新建项目

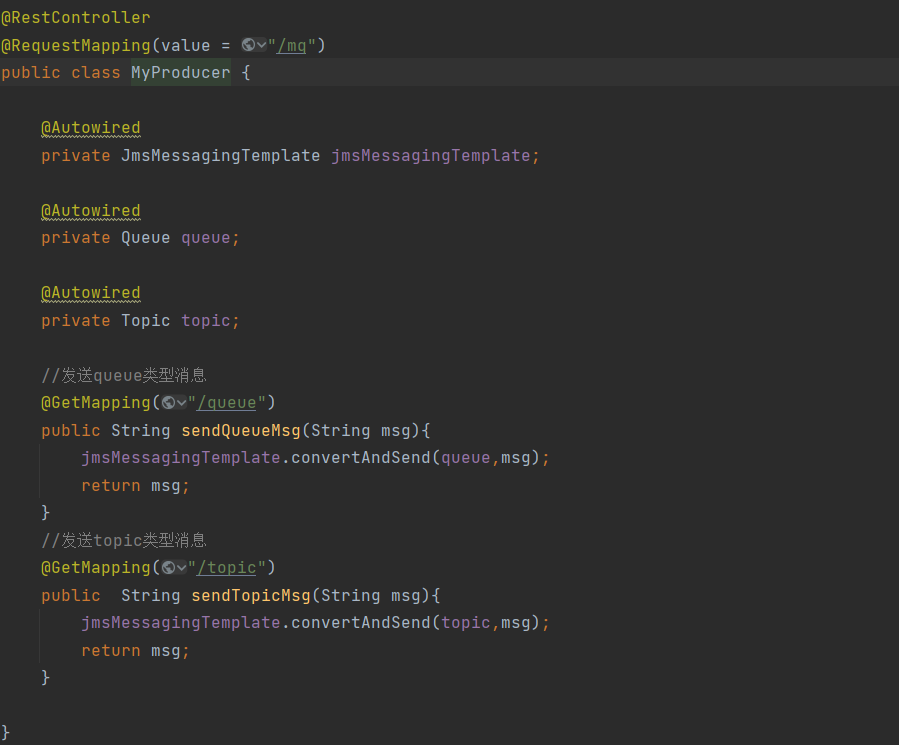
### 配置yml,mqconfig



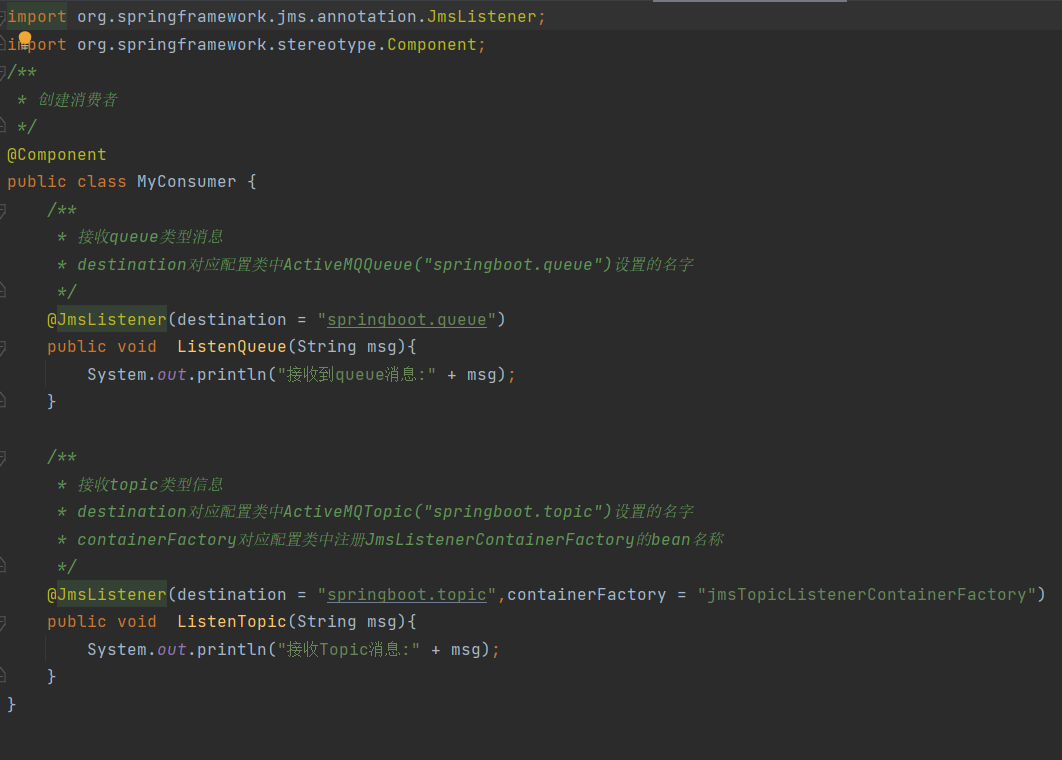
3、导入依赖



4、新建生产者



5、新建消费者



6、登录activemq



# 第五次作业 流程引擎

## 需求场景

了解为什么引进流程引擎。

1、什么是流程引擎。

流程引擎说白了就是用来驱动业务按照我设定的固定流程去流转的东西，在复杂多变的业务情况下，使用既定的流程能够大大降低我们设计业务的成本，并且保证了我们业务执行的准确性。

2、为什么需要学习流程引擎？

1）流程引擎在互联网公司快速盛行，不可或缺

2）流程引擎是java中高级工程师进阶利器

3）掌握流程引擎技术可以提升技术架构和业务建模能力

4）为你的技能栈填上重重的一笔，有效提高你大型互联网公司面试的资本

流程引擎对业务和开发可以起到非常良好的作用：可以快速响应、灵活调整线上流程；业务和开发基于流程模型进行沟通，基于业务建模快速部署；流程可视化，方便查看流程的运行进展

流程机制对团队的作用：提高效率，减少等待；规范行为，落实制度；协同内外，快速响应；监控全面，提升执行。

## 解答方案

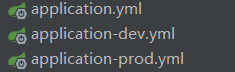
1. 新建module
2. 建立包名
3. 将下载的文件夹下的pom文件中properties、dependencyManagement、dependencies部分copy到自己的pom文件中
4. 将resource中application.yml拷贝到自己module中
5. 将springboot项目入口文件DemoApplication  copy到自己项目中
6. 启动
7. 访问：url为localhost:8080,用户名和密码都为admin

## 解答

### 下载，导入项目。

### 修改配置文件。

### 找到资源文件下的这三个文件。



### 开发环境配置。



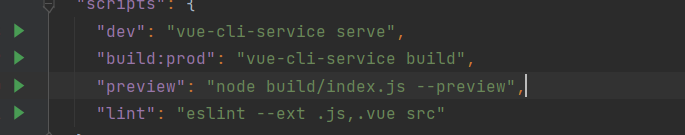




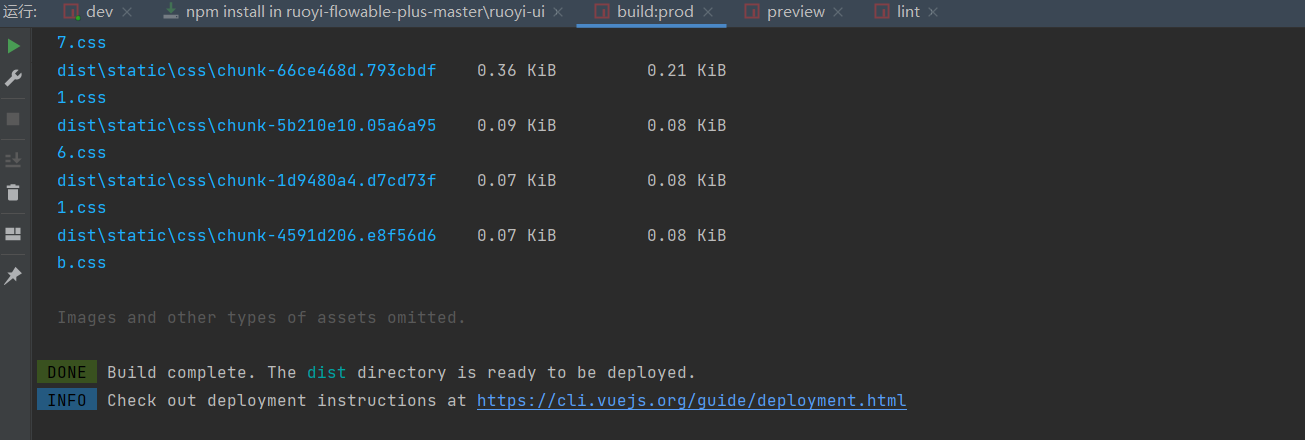
### 启动本机mysql服务

建立配置文件中所写的两个数据库ry-flowable-plus和ry-vue

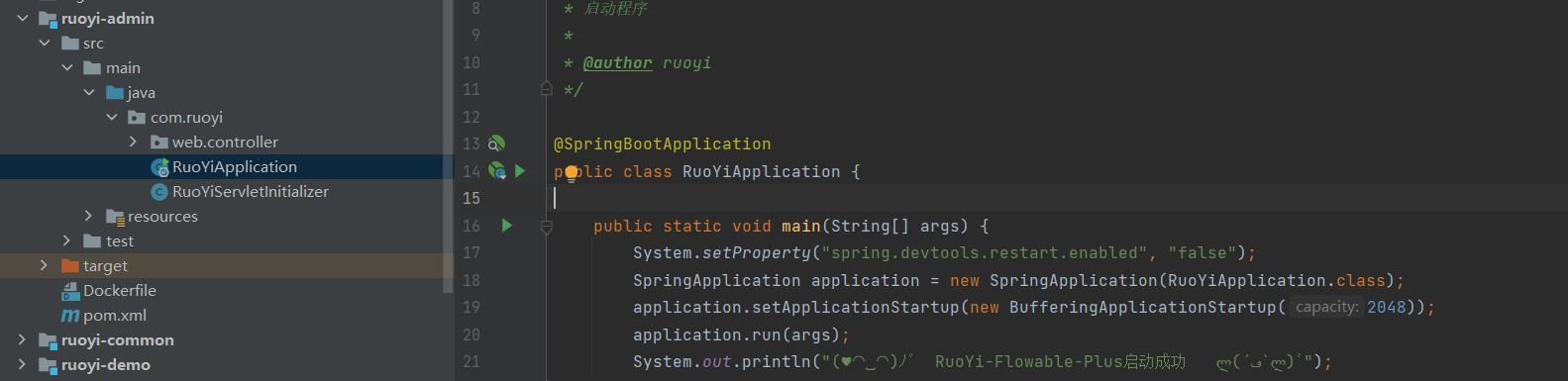
### 找到ui包里的pagejson文件，启动前台。



### 控制台可以看到已启动服务。



### 启动项目。



### 登录若依



# 第六次作业 JDK Timer调度

## 需求场景

Timer是一个用于执行定时任务的类，可以单次执行或按指定时间间隔循环执行（直到主动cancel或线程被杀掉）。Timer中任务处理采用了生产者-消费者模型的设计思想。

## 解答方案

1、创建组件一：TaskQueue（任务队列）

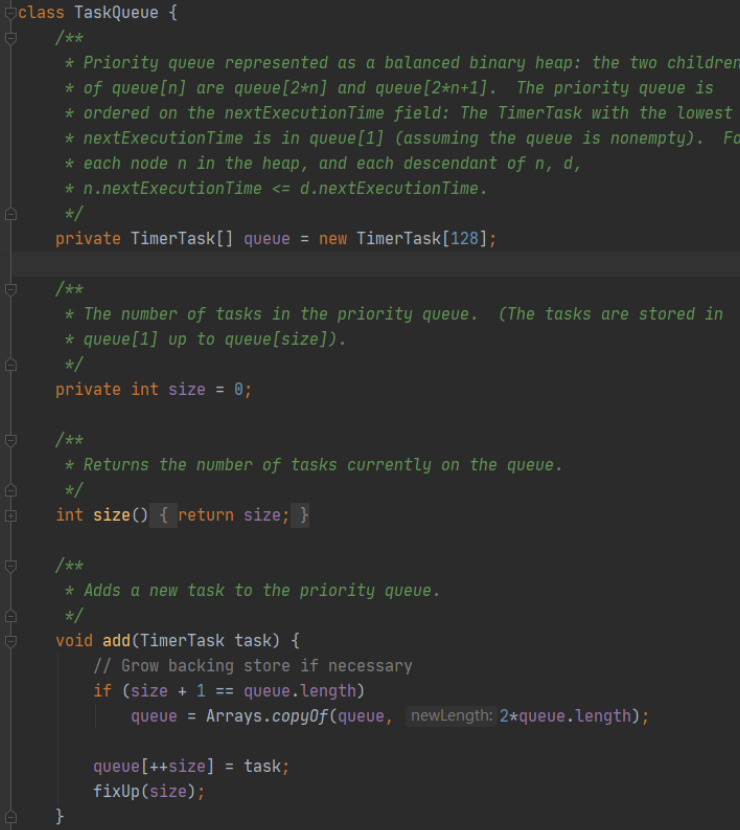
2、创建组件二：TimerTask（任务实体）

3、创建组件三：TimerThread（事件消费者）

## 解答

### TaskQueue：任务队列

一个存储任务的任务池TaskQueue，包含一个初始大小为128的TimerTask数组，负责任务的存储（add）、排序（fixUp、fixDown）、取出(getMin)、清理（removeMin、quickRemove）、循环任务处理（rescheduleMin）以及一些其他基本操作。并通过排序保证队头任务的执行一定是最早的。



### TimerTask：任务实体

TimerTask是任务实体，Runnable接口的实现类。

内部包含用于线程安全的锁lock、用于标记任务状态的字段state、以及一个供用户实现的任务内容抽象方法run().



### TimerThread：事件消费者

一个作为事件消费者的TimerThread，TimerThread中不断获取当前任务队列的队头任务，执行任务。

private TaskQueue queue ;

TimerThread(TaskQueue queue){

this.queue = queue;

}

public void run(){

try{

mainLoop();

}finally{

synchronized(queue){

newTasksMayBeScheduled = false;

queue.clear();

}

}

### 测试代码

public class ScheduleExecutorTest

private void scheduledTask(){

ScheduledExecutorService ses = Executors newScheduledThreadPool( corePoolSize: 5)；

ses.scheduleAtFixedRate(() -> System.out.println("[JavaTask] ScheduledExecutorService-1," + new Date()),initialDelay: 0, period: 5, TimeUnit. SECONDS);

ses.scheduleWithFixedDelay(() -> System.out.println("[JavaTask ScheduledExecutorService-2," + new Date()),initialDelay.p, delay: 3, TimeUnit . SECONDS);

ses.scheduleAtF ixedRate(new TestRunnable( name: "test1"), itiaDelay: 0, period: 1, TimeUnit. SECONDS);

}

static class TestRunnable implements Runnablef{

private String name;

public TestRunnable(String name){this.name = name;}

@0verride

public void run(){System. out .println(LocalDateTim .now()+"--run--” + name);}

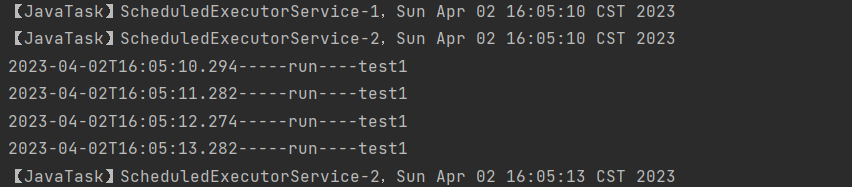
}

public static void nain(String[] args){ new ScheduleExecutorTest().scheduledTaskO;}

}

}

### 测试结果



# 第七次作业 Spring IOC容器

## 需求场景

我们日常的 java 项目开发都是由两个或多个类的彼此合作来实现业务逻辑的，这使得每个对象都需要与其合作的对象的引用（称为所依赖的对象），如果合作的对象的引用或依赖关系由具体的对象来实现，这对复杂的面向对象系统的设计与开发是非常不利的，由此，我们如果能把这些依赖关系和对象的注入交给框架来实现，让具体对象交出手中对于依赖对象的控制，那么就能很大程度上解耦代码，这显然是极有价值的。而这，就是“依赖反转”，即反转对依赖的控制，把控制权从具体的对象中转交到平台或者框架。

实现依赖反转的实现有很多种，在 Spring 中，IoC 容器就是实现这个模式的载体，它可以在对象生成或初始化的过程中，直接将数据或者依赖对象的引用注入到对象的数据域中从而来实现方法调用的依赖。而这种依赖注入是递归的，依赖对象会被逐层注入，从而建立起一套有序的对象依赖关系，简化了对象依赖关系的管理，把面向对象过程中需要执行的如何新建对象、为对象引用赋值等操作交由容器统一管理，极大程度上减低了面向对象编程的复杂性。

在 Spring 中，Spring IoC 提供了一个基本 JavaBean 容器，通过 IoC 模式管理依赖关系，并通过依赖注入和 AOP 切面增强了为 JavaBean 这样的 POJO 对象提供了事务管理、声明周期管理等功能。

在应用开发中，当我们在设计组件时，往往需要引入和调用其他组件的服务时，这种依赖关系如果固化在组件设计中就会导致组件之间的耦合和维护难度的增大，这个时候如果使用 IoC 容器，把资源获取的方式反转，让 IoC 容器主动管理这些依赖关系，将依赖关系注入到组件中，那么这些依赖关系的适配和管理就会更加灵活。

在应用管理依赖关系时，如果在 IOC 实现依赖反转的过程中，能通过可视化的文本来完成配置，并且通过工具对这些配置信息进行可视化的管理和浏览，那么肯定能提高依赖关系的管理水平，而且如果耦合关系变动，并不需要重新修改和编译 Java 代码，这符合在面向对象过程中的开闭原则。

## 解答方案

定义元数据

1、Controller层的标注@Controller

2、Service层的标注@Service

3、Autowired依赖注入注解@Autowired

必备工具类

1、配置文件读写工具

2、反射工具类

3、属性文件工具类

4、类操作工具类

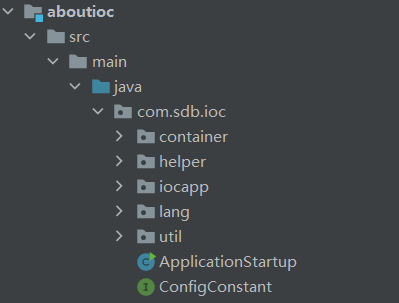
其他部分

1、class集合

2、bean集合

## 解答

### 新建项目。



### 修改pom文件。

添加如下依赖。

<dependencies>

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-lang3</artifactId>

<version>3.4</version>

</dependency>

<dependency>

<groupId>org.apache.commons</groupId>

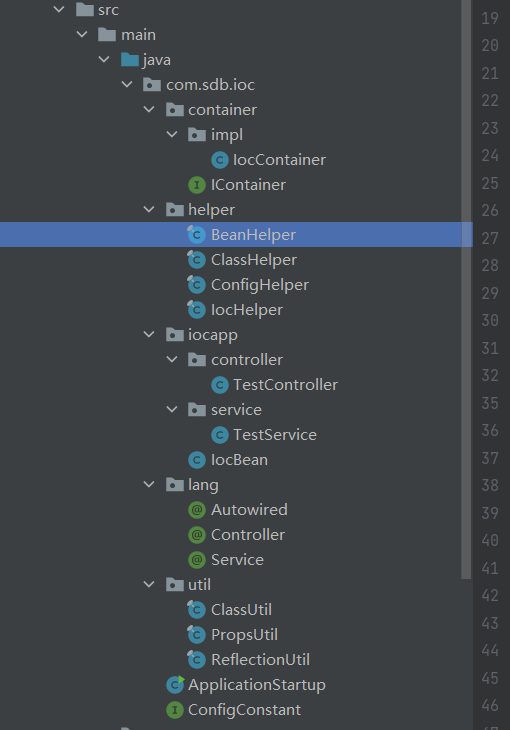
<artifactId>commons-collections4</artifactId

<version>4.4</ version>

</dependency>

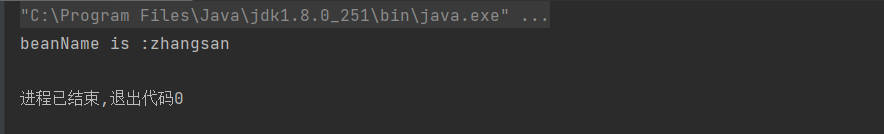
### 导入代码。

分别导入，impl,bean容器，controller，service，工具类，启动类。



### 测试

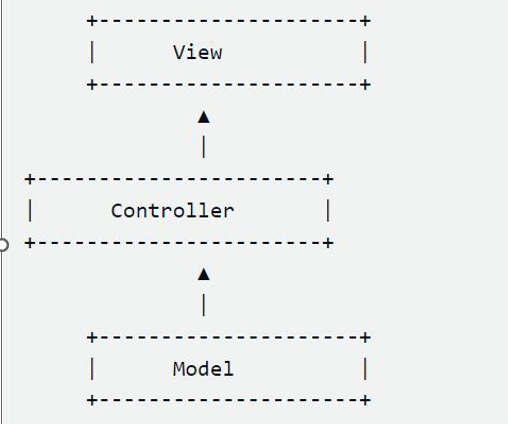
运行启动类出现测试结果



# 第八次作业 MVC架构

## 需求场景

MVC (Model-View-Controller）是一种软件架构模式，它将应用程序分为三个主要部分:模型(Model)、视图 (View)和控制器(Controller)。以下是MVC架构图:



在MVC中，应用程序的数据和业务逻辑都包含在模型中。视图是用户界面的可见部分，它显示模型中的数据并允许用户与数据进行交互。控制器协调模型和视图之间的交互,处理用户输入并更新模型和视图。控制器还可以接收来自其他控制器或外部系统的事件,并相应地更新模型和视图。

## 解答方案

1、Model层表示数据模型，它具有一个值属性和相应的getter和setter方法。

2、View层表示用户界面，它具有一个显示值的方法和一个获取用户输入的方法。

3、Controller层表示控制器，它协调模型和视图之间的交互，处理用户输入并更新模型和视图。

4、Main类包含程序的入口点，它创建Model、View和Controller对象，并调用Controller的processUserInput方法来处理用户输入并更新视图。

## 解答

### 新建MVC模块

### 添加依赖项

<dependencies>

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-lang3</artifactId>

<version>3.4</version>

</dependency>

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-lang3</artifactId>

<version>3.4</version>

</dependency>

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-collections4</artifactId

<version>4.4</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>2.5</version>

</dependency>

</dependencies>

### 新建包名

1.Controller层（控制层）

2.Model层（数据层）

3.View层（视图层）

### 添加代码

1. Controller层（控制层）

public class UserController{

private UserModel userNodel;

private UserView userView;

public UserController (UserModel userModel, UserView userView){ this.userModel : userModel;

this.userView : userView;

}

public void setUserName(String name){

userModel. setName(name);

}

public void setUserAge(int age){

userModel. setAge(age);

}

public String getUserName(){

return userModel. getName();}

public int getUserAge(){

return userModel, getAge();}

public void updateView(){

userView,printUserDetails(userModel. getName(), userModel, getAge());}

}

控制数据层和视图层，进行数据交互。获得数据层用户数据向视图层传递。

1. Model层（数据层）

publlc class UserModel{

private String name;

private int age;

public String getName(){

return name;

}

public void setName (String name){

this.name.name;

}

public int getAge(){

return age;

}

public void setAge(int age){

This.age = age;

}

}

存放数据

3.View层（视图层）

public class UserView{

public void printUserDetails(String name, int age){

System.out.println("Name: " + name);

System.out.println("Age: " + age);

输出数据

4.Main启动类编写

public class Main{

public static void main(String[] args){

UserModel userModel = new UserModel();

UserView userView = new UserView();

UserController userController = new UserController(userModel, userView);

userController.setUserName("张三");

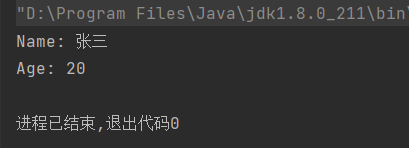
userController.setUserAge(20);

userController.updateView();

}

}

### 测试结果



# 第九次作业 SpringAOP

## 需求场景

springAOP解决了除了核心逻辑，还要关注非核心逻辑,代码重复率高（比如输出同样的日志格式）的问题。springAOP把非核心业务封装起来，只写核心业务,将公共非核心的封装起来，格式在公共部分给出即可。

springAOP用法

1、权限管理

情景1：控制用户的功能权限

方案详述：在@ControllerAdvice里边，处理全局请求，控制权限。

权限管理的其他方案：（除了AOP之外的方案）

在过滤器或者拦截器中处理

使用Shiro中间件

2、异常处理

情景1：在@ControllerAdvice里边，处理全局异常

情景2：将Dubbo接口作为切面，统一处理Dubbo接口里边的异常

3、操作日志

情景1：按产品的需求，有的接口需要记录操作日志

自定义注解，需要记录操作日志的，则在Controller的方法上加此注解

AOP中判断，如果有这个自定义注解，则将参数异步写到日志数据库

4、将数据同步到ES

情景1：增删改数据时，同时要处理MySQL和ES

将相关类作为切面，若数据库提交，则写到ES；若回滚，则不写到ES

5、事务控制

情景1：使用Spring的@Transactional

## 解答方案

1、创建切面代理，抽象类

2、创建代理接口

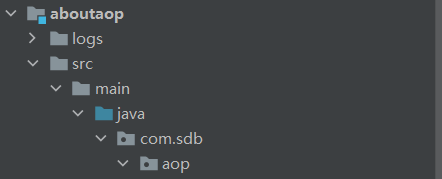
3、创建代理链

4、创建代理管理器

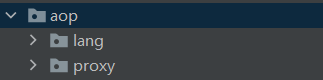
5、创建事务代理

## 解答

### 创建aop包



### 在aop包下创建，注解包lang,代理包proxy



### 创建切面代理抽象类

package com.sdb.aop.proxy;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import java.lang.reflect.Method;

/\*\*

\* 切面代理 抽象类

\*/

public abstract class AspectProxy implements Proxy {

private static final Logger logger = LoggerFactory.getLogger(AspectProxy.class);

@Override

public final Object doProxy(ProxyChain proxyChain) throws Throwable {

Object result = null;

Class<?> cls = proxyChain.getTargetClass();

Method method = proxyChain.getTargetMethod();

Object[] params = proxyChain.getMethodParams();

begin();

try {

if (intercept(method, params)) {

before(method, params);

result = proxyChain.doProxyChain();

after(method, params);

} else {

result = proxyChain.doProxyChain();

}

} catch (Exception e) {

logger.error("proxy failure", e);

System.out.println("proxy failure");

error(method, params, e);

throw e;

} finally {

end();

}

return result;

}

/\*\*

\* 开始增强

\*/

public void begin() {

}

/\*\*

\* 切入点判断

\*/

public boolean intercept(Method method, Object[] params) throws Throwable {

return true;

}

/\*\*

\* 前置增强

\*/

public void before(Method method, Object[] params) throws Throwable {

}

/\*\*

\* 后置增强

\*/

public void after(Method method, Object[] params) throws Throwable {

}

/\*\*

\* 异常增强

\*/

public void error(Method method, Object[] params, Throwable e) {

}

/\*\*

\* 最终增强

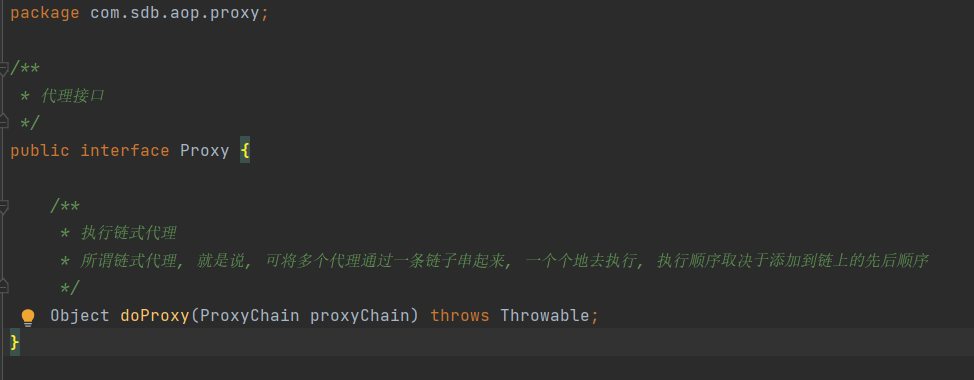
\*/

public void end() {

}

}

### 创建代理接口



### 创建代理链

package com.sdb.aop.proxy;

import net.sf.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* 代理链

\*/

public class ProxyChain {

private final Class<?> targetClass; //目标类

private final Object targetObject; //目标对象

private final Method targetMethod; //目标方法

private final MethodProxy methodProxy; //方法代理

private final Object[] methodParams; //方法参数

private List<Proxy> proxyList = new ArrayList<>(); //代理列表

private int proxyIndex = 0; //代理索引

public ProxyChain(Class<?> targetClass, Object targetObject, Method targetMethod, MethodProxy methodProxy, Object[] methodParams, List<Proxy> proxyList) {

this.targetClass = targetClass;

this.targetObject = targetObject;

this.targetMethod = targetMethod;

this.methodProxy = methodProxy;

this.methodParams = methodParams;

this.proxyList = proxyList;

}

public Object[] getMethodParams() {

return methodParams;

}

public Class<?> getTargetClass() {

return targetClass;

}

public Method getTargetMethod() {

return targetMethod;

}

/\*\*

\* 递归执行

\*/

public Object doProxyChain() throws Throwable {

Object methodResult;

if (proxyIndex < proxyList.size()) {

//执行增强方法

methodResult = proxyList.get(proxyIndex++).doProxy(this);

} else {

//目标方法最后执行且只执行一次

methodResult = methodProxy.invokeSuper(targetObject, methodParams);

}

return methodResult;

}

}

### 创建代理管理器

package com.sdb.aop.proxy;

import net.sf.cglib.proxy.Enhancer;

import net.sf.cglib.proxy.MethodInterceptor;

import net.sf.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

import java.util.List;

/\*\*

\* 代理管理器

\*/

public class ProxyFactory {

/\*\*

\* 输入一个目标类和一组Proxy接口实现, 输出一个代理对象

\*/

@SuppressWarnings("unchecked")

public static <T> T createProxy(final Class<?> targetClass, final List<Proxy> proxyList) {

return (T) Enhancer.create(targetClass, new MethodInterceptor() {

/\*\*

\* 代理方法, 每次调用目标方法时都会先创建一个 ProxyChain 对象, 然后调用该对象的 doProxyChain() 方法.

\*/

@Override

public Object intercept(Object targetObject, Method targetMethod, Object[] methodParams, MethodProxy methodProxy) throws Throwable {

return new ProxyChain(targetClass, targetObject, targetMethod, methodProxy, methodParams, proxyList).doProxyChain();

}

});

}

}

### 创建事务代理

package com.sdb.aop.proxy;

import com.sdb.aop.proxy.helper.DatabaseHelper;

import com.sdb.aop.lang.Transactional;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import java.lang.reflect.Method;

/\*\*

\* 事务代理

\*/

public class TransactionProxy implements Proxy {

private static final Logger LOGGER = LoggerFactory.getLogger(TransactionProxy.class);

@Override

public Object doProxy(ProxyChain proxyChain) throws Throwable {

Object result;

Method method = proxyChain.getTargetMethod();

//加了@Transactional注解的方法要做事务处理

if (method.isAnnotationPresent(Transactional.class)) {

try {

DatabaseHelper.beginTransaction();

System.out.println("begin transaction");

LOGGER.debug("begin transaction");

result = proxyChain.doProxyChain();

DatabaseHelper.commitTransaction();

System.out.println("commit transaction");

LOGGER.debug("commit transaction");

} catch (Exception e) {

DatabaseHelper.rollbackTransaction();

LOGGER.debug("rollback transaction");

throw e;

}

} else {

result = proxyChain.doProxyChain();

}

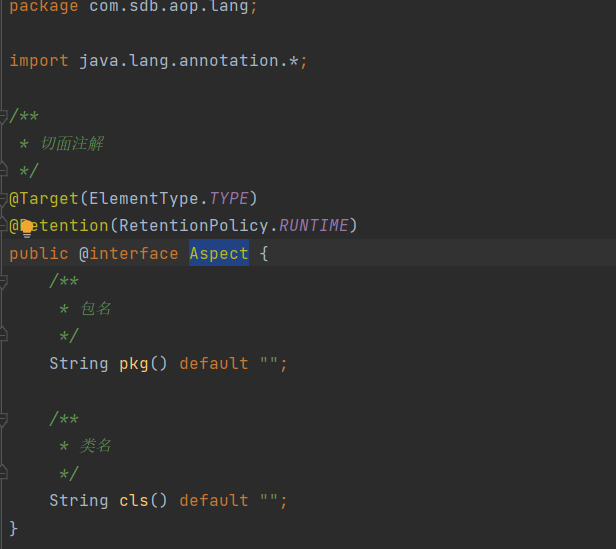
return result;

}

}

### 编写lang注解类。

1.Aspect注解



2.Transactional注解

