**1.环境准备**

**1.1 安装Docker**

下面的命令一起执行，安装 gcc、gcc-c+ 等环节 ，以及安装 docker-ce

yum -y install gcc  
yum -y install gcc-c++  
yum install -y yum-utils device-mapper-persistent-data lvm2  
yum-config-manager --add-repo http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo  
yum makecache fast  
yum -y install docker-ce  
systemctl start docker  
​

配置国内仓库

创建目录

mkdir -p /etc/docker

编辑配置

vim /etc/docker/daemon.json

在文件中添加以下配置

{  
"registry-mirrors": ["https://docker.mirrors.ustc.edu.cn/","https://hub-mirror.c.163.com","https://registry.docker-cn.com"]  
}

加载配置文件以及重启docker

systemctl daemon-reload  
systemctl restart docker  
​

**1.2 安装 Redis**

docker run -di --name pochi\_redis -p 6379:6379 redis

**1.3 安装 MySQL**

在本地创建mysql的映射目录，用来存放数据、日志、配置文件

mkdir -p /usr/local/docker/mysql/data /usr/local/docker/mysql/logs /usr/local/docker/mysql/conf

在/usr/local/docker/mysql/conf 下编辑配置文件

vi my.cnf

内容如下

**[client]**  
default-character-set=utf8mb4  
**[mysql]**  
default-character-set=utf8mb4  
**[mysqld]**  
character-set-server=utf8mb4  
default-time\_zone = '+8:00'

[mysqld]

skip-name-resolve

执行命令安装

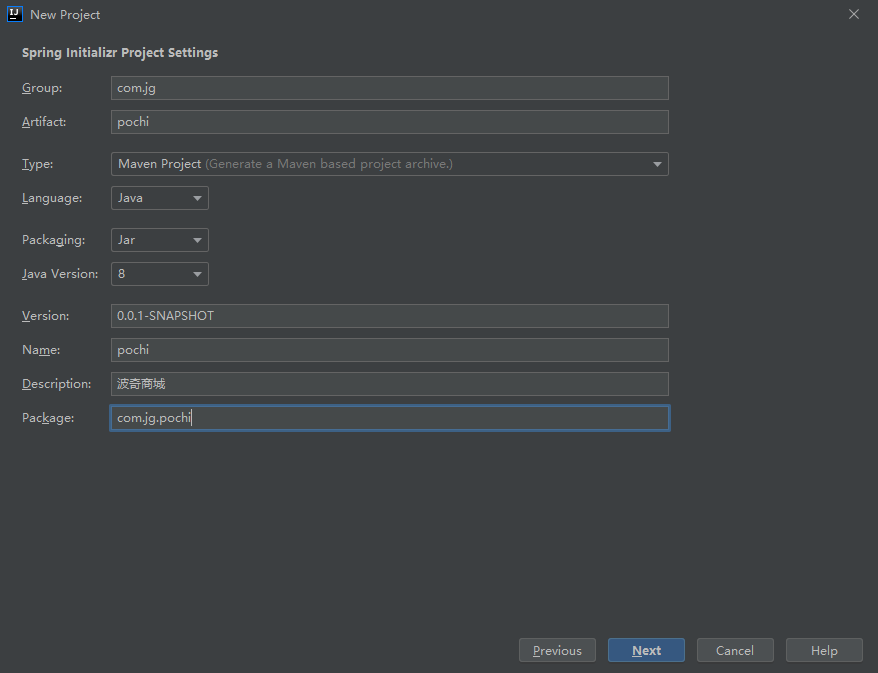
docker run -di --name mysql -p 3306:3306 \  
--network host \  
-v /usr/local/docker/mysql/conf:/etc/mysql/conf.d \  
-v /usr/local/docker/mysql/logs:/logs \  
-v /usr/local/docker/mysql/data:/var/lib/mysql \  
-e MYSQL\_ROOT\_PASSWORD=yangdeshi -d mysql:5.7  
​

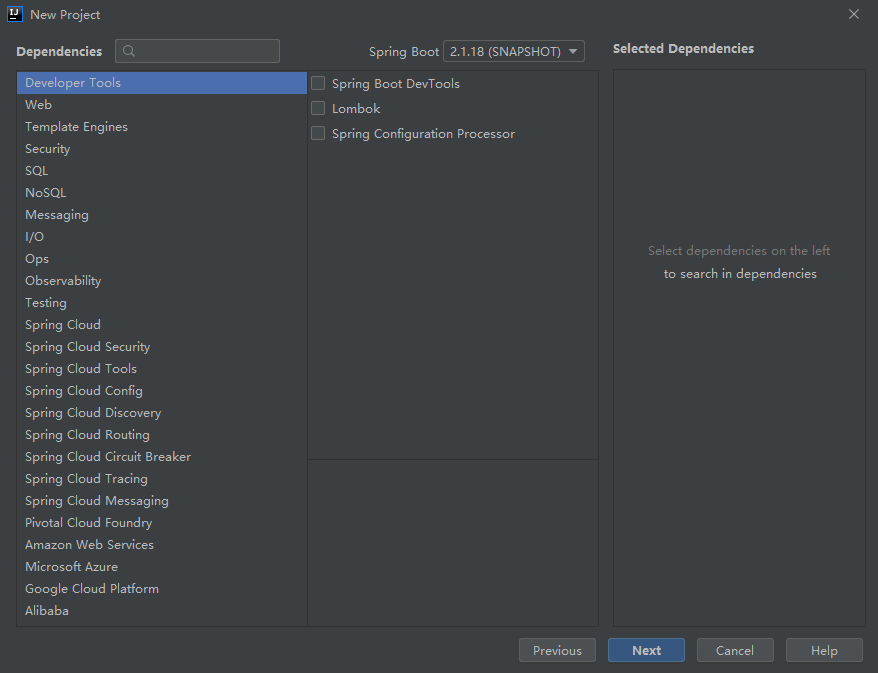
**1.4 导入数据库文件**

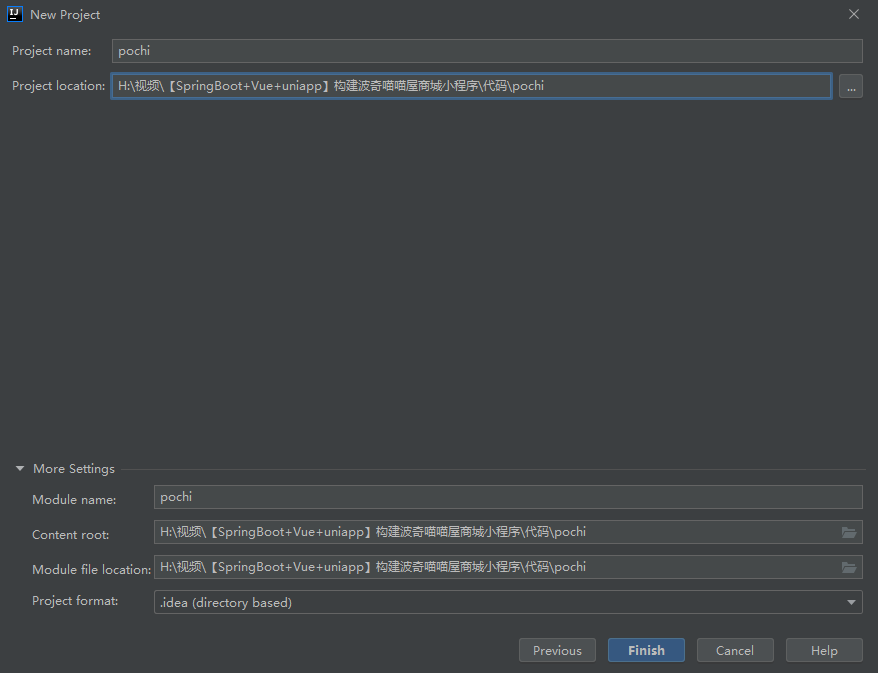
新建数据库 pochi，导入资料下的 pochi.sql 文件

**2. 后端搭建**

**2.1 创建项目**







**2.2 pom.xml配置**

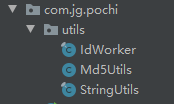
<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">  
   <modelVersion>4.0.0</modelVersion>  
   <parent>  
       <groupId>org.springframework.boot</groupId>  
       <artifactId>spring-boot-starter-parent</artifactId>  
       <version>2.3.3.RELEASE</version>  
   </parent>  
​  
   <groupId>com.jg</groupId>  
   <artifactId>pochi</artifactId>  
   <version>0.0.1-SNAPSHOT</version>  
   <name>pochi</name>  
   <description>波奇商城</description>  
​  
   <properties>  
       <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
       <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  
       <maven.compiler.encoding>UTF-8</maven.compiler.encoding>  
       <maven.compiler.source>1.8</maven.compiler.source>  
       <maven.compiler.target>1.8</maven.compiler.target>  
       <mybatis.starter.version>1.3.2</mybatis.starter.version>  
       <jackson.version>2.11.2</jackson.version>  
       <java.version>1.8</java.version>  
       <skipTests>true</skipTests>  
   </properties>  
​  
   <dependencies>  
       <!--commons -->  
       <dependency>  
           <groupId>org.apache.commons</groupId>  
           <artifactId>commons-lang3</artifactId>  
           <version>3.6</version>  
       </dependency>  
       <dependency>  
           <groupId>commons-configuration</groupId>  
           <artifactId>commons-configuration</artifactId>  
           <version>1.10</version>  
       </dependency>  
       <dependency>  
           <groupId>commons-io</groupId>  
           <artifactId>commons-io</artifactId>  
           <version>2.5</version>  
       </dependency>  
​  
​  
       <dependency>  
           <groupId>junit</groupId>  
           <artifactId>junit</artifactId>  
           <scope>test</scope>  
       </dependency>  
​  
       <dependency>  
           <groupId>org.projectlombok</groupId>  
           <artifactId>lombok</artifactId>  
       </dependency>  
​  
       <dependency>  
           <groupId>org.jsoup</groupId>  
           <artifactId>jsoup</artifactId>  
           <version>1.11.3</version>  
       </dependency>  
       <dependency>  
           <groupId>com.alibaba</groupId>  
           <artifactId>fastjson</artifactId>  
           <version>1.2.69</version>  
       </dependency>  
​  
       <!-- JSON处理插件。json中的null值将不会回显给前端 -->  
       <dependency>  
           <groupId>com.fasterxml.jackson.core</groupId>  
           <artifactId>jackson-databind</artifactId>  
           <version>${jackson.version}</version>  
       </dependency>  
       <dependency>  
           <groupId>com.fasterxml.jackson.core</groupId>  
           <artifactId>jackson-annotations</artifactId>  
           <version>${jackson.version}</version>  
       </dependency>  
       <dependency>  
           <groupId>com.fasterxml.jackson.core</groupId>  
           <artifactId>jackson-core</artifactId>  
           <version>${jackson.version}</version>  
       </dependency>  
       <dependency>  
           <groupId>org.springframework.boot</groupId>  
           <artifactId>spring-boot-starter-data-redis</artifactId>  
       </dependency>  
       <dependency>  
           <groupId>org.springframework.boot</groupId>  
           <artifactId>spring-boot-starter-web</artifactId>  
       </dependency>  
​  
       <dependency>  
           <groupId>org.springframework.boot</groupId>  
           <artifactId>spring-boot-starter-test</artifactId>  
           <scope>test</scope>  
       </dependency>  
       <dependency>  
           <groupId>org.springframework.boot</groupId>  
           <artifactId>spring-boot-starter-aop</artifactId>  
       </dependency>  
       <dependency>  
           <groupId>mysql</groupId>  
           <artifactId>mysql-connector-java</artifactId>  
       </dependency>  
​  
       <dependency>  
           <groupId>org.mybatis.spring.boot</groupId>  
           <artifactId>mybatis-spring-boot-starter</artifactId>  
           <version>${mybatis.starter.version}</version>  
       </dependency>  
​  
       <dependency>  
           <groupId>org.apache.shiro</groupId>  
           <artifactId>shiro-spring</artifactId>  
           <version>1.3.2</version>  
       </dependency>  
       <dependency>  
           <groupId>com.alibaba</groupId>  
           <artifactId>druid-spring-boot-starter</artifactId>  
           <version>1.1.10</version>  
       </dependency>  
       <dependency>  
           <groupId>com.google.guava</groupId>  
           <artifactId>guava</artifactId>  
           <version>16.0</version>  
       </dependency>  
       <dependency>  
           <groupId>com.github.jsqlparser</groupId>  
           <artifactId>jsqlparser</artifactId>  
           <version>1.3</version>  
       </dependency>  
​  
       <dependency>  
           <groupId>org.springframework.boot</groupId>  
           <artifactId>spring-boot-starter-data-mongodb</artifactId>  
       </dependency>  
​  
       <dependency>  
           <groupId>com.aliyun.oss</groupId>  
           <artifactId>aliyun-sdk-oss</artifactId>  
           <version>3.8.0</version>  
       </dependency>  
       <dependency>  
           <groupId>org.springframework.boot</groupId>  
           <artifactId>spring-boot-starter-data-elasticsearch</artifactId>  
       </dependency>  
   </dependencies>  
​  
   <build>  
       <plugins>  
           <plugin>  
               <groupId>org.springframework.boot</groupId>  
               <artifactId>spring-boot-maven-plugin</artifactId>  
           </plugin>  
       </plugins>  
   </build>  
​  
</project>  
​

**2.3 application.yml配置**

spring:  
datasource:  
  druid:  
    url: jdbc:mysql://localhost:3306/pochi?serverTimezone=GMT%2B8&useUnicode=true&characterEncoding=utf8  
    username: root  
    password: yangdeshi  
    driver-class-name: com.mysql.jdbc.Driver  
    initialSize: 5  
    minIdle: 5  
    maxActive: 20  
    maxWait: 10000  
    testOnBorrow: true  
    testOnReturn: false  
    timeBetweenEvictionRunsMillis: 60000  
    minEvictableIdleTimeMillis: 300000  
servlet:  
  multipart:  
    max-file-size: 50MB  
    max-request-size: 100MB  
redis:  
  host: 127.0.0.1  
  port: 6379  
mybatis:  
mapper-locations: classpath:mapper/\*Mapper.xml  
type-aliases-package: com.jg.pochi  
​

**2.4 导入工具类**

新建 utils 包，将资料中的工具类导入进去



**2.5 编写分页和返回集类**

放入 utils 包下

**2.5.1 Page**

package com.jg.pochi.common;  
  
import java.io.Serializable;  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
  
*/\*\*  
 \* 分页类  
 \* Author Peekaboo  
 \** ***@Date*** *2021/11/19 17:36  
 \*/*public class Page*<*T*>* implements Serializable *{* //当前页数  
 private Integer pageNumber;  
  
 //每页显示条数  
 private Integer pageSize;  
  
 //总条数  
 private Integer totalCount;  
  
 //总页数  
 private Integer totalPage;  
  
 //数据  
 private List*<*T*>* list;  
  
 */\*\*  
 \* 传参数据  
 \* 由于传参不固定因此用Map，其他情况一般不随便用  
 \*/* private Map*<*String, Object*>* params = new HashMap*<>(*8*)*;  
  
 */\*\*  
 \* 排序列  
 \*/* private String sortColumn;  
  
 */\*\*  
 \* 排序方式，asc或者desc  
 \*/* private String sortMethod;  
  
 public Integer getIndex*(){* return *(*pageNumber - 1*)* \* pageSize;  
 *}  
  
 /\*\*  
 \* 在设置总条数时，计算总页数  
 \*/* public void setTotalCount*(*Integer totalCount*){* this.totalCount = totalCount;  
 this.totalPage = *(*int*)*Math.*ceil(*totalCount \* 1.0 / pageSize*)*;  
 *}  
}*

**2.5.2 Result**

package com.jg.pochi.utils;  
​  
import com.jg.pochi.enums.ResultEnum;  
import lombok.Data;  
​  
import java.io.Serializable;  
​  
/\*\*  
\* 统一返回结果  
\*  
\* @Author: 杨德石  
\* @Date: 2020/2/9 14:26  
\* @Version 1.0  
\*/  
@Data  
public class Result<T> implements Serializable {  
​  
   private Integer code;  
   private String msg;  
   private T data;  
​  
   public Result() {  
       this.code = ResultEnum.SUCCESS.getCode();  
       this.msg = ResultEnum.SUCCESS.getMsg();  
  }  
​  
   public Result(String msg) {  
       this.code = ResultEnum.SUCCESS.getCode();  
       this.msg = msg;  
  }  
​  
   public Result(T data) {  
       this.code = ResultEnum.SUCCESS.getCode();  
       this.msg = ResultEnum.SUCCESS.getMsg();  
       this.data = data;  
  }  
​  
   public Result(String msg, T data) {  
       this.code = ResultEnum.SUCCESS.getCode();  
       this.msg = msg;  
       this.data = data;  
  }  
​  
   public Result(Integer code, String msg) {  
       this.code = code;  
       this.msg = msg;  
  }  
​  
   public Result(ResultEnum resultEnum, String msg) {  
       this.code = resultEnum.getCode();  
       this.msg = msg;  
  }  
​  
   public Result(ResultEnum resultEnum) {  
       this.code = resultEnum.getCode();  
       this.msg = resultEnum.getMsg();  
  }  
​  
   public Result(Integer code, String msg, T data) {  
       this.code = code;  
       this.msg = msg;  
       this.data = data;  
  }  
}  
​

**2.6 返回码枚举**

项目下创建 enums 包

package com.jg.pochi.enums;  
​  
import lombok.Getter;  
​  
/\*\*  
\* 返回结果枚举  
\* @Author: 杨德石  
\* @Date: 2020/2/9 14:16  
\* @Version 1.0  
\*/  
@Getter  
public enum ResultEnum {  
   /\*\*  
    \* 返回结果枚举，每个枚举代表着一个返回状态  
    \*/  
   SUCCESS(20000, "操作成功！"),  
   ERROR(40000, "操作失败！"),  
   DATA\_NOT\_FOUND(40001, "查询失败！"),  
   PARAMS\_NULL(40002, "参数不能为空！"),  
   PARAMS\_ERROR(40005, "参数不合法！"),  
​  
   NOT\_LOGIN(40003, "当前账号未登录！")  
  ;  
   private Integer code;  
   private String msg;  
​  
   ResultEnum(Integer code, String msg) {  
       this.code = code;  
       this.msg = msg;  
  }  
}  
​

**2.7 自定义异常**

新建 exception 包

package com.jg.pochi.exception;  
​  
import com.jg.pochi.enums.ResultEnum;  
​  
/\*\*  
\* 自定义异常  
\* @author yds  
\* @date  
\*/  
public class PochiException extends RuntimeException {  
​  
   private static final long serialVersionUID = 2450214686001409867L;  
​  
   private Integer errorCode = ResultEnum.ERROR.getCode();  
​  
   public PochiException(ResultEnum resultEnum) {  
       super(resultEnum.getMsg());  
       this.errorCode = resultEnum.getCode();  
  }  
​  
   public PochiException(ResultEnum resultEnum, Throwable throwable) {  
       super(resultEnum.getMsg(), throwable);  
       this.errorCode = resultEnum.getCode();  
  }  
​  
   public PochiException(Integer errorCode, String msg) {  
       super(msg);  
       this.errorCode = errorCode;  
  }  
​  
   public PochiException(String msg) {  
       super(msg);  
  }  
​  
   public PochiException(Throwable throwable) {  
       super(throwable);  
  }  
​  
   public PochiException(String msg, Throwable throwable) {  
       super(msg, throwable);  
  }  
​  
   public Integer getErrorCode() {  
       return errorCode;  
  }  
}  
​

**2.8 统一异常处理**

知识备注：

使用 @ControllerAdvice 实现全局异常处理，只需要定义类，添加该注解即可定义方式如下：

@ControllerAdvice

public class MyGlobalExceptionHandler {

@ExceptionHandler(Exception.class)

public ModelAndView customException(Exception e) {

ModelAndView mv = new ModelAndView();

mv.addObject("message", e.getMessage());

mv.setViewName("myerror");

return mv;

}

}

在该类中，可以定义多个方法，不同的方法处理不同的异常，例如专门处理空指针的方法、专门处理数组越界的方法...，也可以直接向上面代码一样，在一个方法中处理所有的异常信息。

@ExceptionHandler 注解用来指明异常的处理类型，即如果这里指定为 NullpointerException，则数组越界异常就不会进到这个方法中来。

新建 advice 包

package com.jg.pochi.advice;  
​  
import com.jg.pochi.exception.PochiException;  
import com.jg.pochi.utils.Result;  
import lombok.extern.slf4j.Slf4j;  
import org.springframework.web.bind.annotation.ControllerAdvice;  
import org.springframework.web.bind.annotation.ExceptionHandler;  
import org.springframework.web.bind.annotation.ResponseBody;  
​  
/\*\*  
\* 定义统一异常处理  
\*  
\* @Author: 杨德石  
\* @Date: 2020/2/9 14:25  
\* @Version 1.0  
\*/  
@ControllerAdvice  
@Slf4j  
public class PochiExceptionAdvice {  
​  
   /\*\*  
    \* 统一处理 BlogException  
    \*  
    \* @param exception  
    \*/  
   @ExceptionHandler(PochiException.class)  
   @ResponseBody  
   public Result<Object> exceptionHandler(PochiException exception) {  
       log.error("统一异常处理：", exception);  
       return new Result<>(exception.getErrorCode(), exception.getMessage());  
  }  
}  
​

**2.9 处理Long类型精度丢失**

背景：

由于JS没有处理Long类型的能力

在启动类下面加上如下全局处理Long类型

   /\*\*  
    \* 处理Long类型精度丢失  
    \* @return  
    \*/  
   @Bean("jackson2ObjectMapperBuilderCustomizer")  
   public Jackson2ObjectMapperBuilderCustomizer jackson2ObjectMapperBuilderCustomizer() {  
       return jacksonObjectMapperBuilder -> jacksonObjectMapperBuilder.serializerByType(Long.class, ToStringSerializer.instance)  
              .serializerByType(Long.TYPE, ToStringSerializer.instance);  
  }  
​

**2.10 启动类**

@SpringBootApplication  
@EnableTransactionManagement  
@MapperScan("com.jg.pochi.mapper")  
public class PochiApplication {  
​  
   public static void main(String[] args) {  
       SpringApplication.run(PochiApplication.class, args);  
  }  
​  
}  
​

**2.11 状态值枚举**

新建类 StateEnum

package com.jg.pochi.enums;  
​  
import lombok.Getter;  
​  
/\*\*  
\* 状态码枚举。所有的状态码都在这里编写  
\*  
\* @Author: 杨德石  
\* @Date: 2020/2/9 14:19  
\* @Version 1.0  
\*/  
@Getter  
public enum StateEnums {  
   /\*\*  
    \* 逻辑删除状态  
    \*/  
   DELETED(1, "已删除"),  
   NOT\_DELETED(0, "未删除"),  
​  
   /\*\*  
    \* 启用状态  
    \*/  
   ENABLED(1, "启用"),  
   NOT\_ENABLE(0, "未启用"),  
​  
   /\*\*  
    \* 性别状态  
    \*/  
   SEX\_MAN(1, "男"),  
   SEX\_WOMAN(2, "女"),  
​  
   /\*\*  
    \* 请求访问状态枚举  
    \*/  
   REQUEST\_SUCCESS(1, "请求正常"),  
   REQUEST\_ERROR(0, "请求异常"),  
;  
​  
   private Integer code;  
   private String msg;  
​  
   StateEnums(Integer code, String msg) {  
       this.code = code;  
       this.msg = msg;  
  }  
}  
​

**2.12 日志记录**

**2.12.1 日志文件记录**

在资料中找到 logback-spring.xml 文件，复制到项目的 resource 下即可。

### 知识备注：

ERROR 指出虽然发生错误事件，但仍然不影响系统的继续运行。打印错误和异常信息，如果不想输出太多的日志，可以使用这个级别

WARN 表明会出现潜在错误的情形，有些信息不是错误信息，但是也要给程序员的一些提示

INFO 消息在粗粒度级别上突出强调应用程序的运行过程。打印一些你感兴趣的或者重要的信息，这个可以用于生产环境中输出程序运行的一些重要信息，但是不能滥用，避免打印过多的日志。

DEBUG 指出细粒度信息事件对调试应用程序是非常有帮助的，主要用于开发过程中打印一些运行信息。

**2.12.2 创建日志表相关类**

知识备注：

Java中的POJO是什么意思？

POJO（普通普通java对象）是一个简单的java对象，实际上是一个常见的JavaBeans。创建它是为了避免与EJB混淆。

1. 它通常指不使用实体bean的普通Java对象，POJO可以用作支持业务逻辑的辅助类。

2. 本质上，POJO可以理解为一个简单的实体类。顾名思义，POJO类的作用是方便程序员使用数据库中的数据表。对于大多数程序员来说，使用POJO类作为对象是非常方便的。当然，调用其get和set方法也很方便。POJO类也给Struts框架的配置带来了极大的方便。

POJO和JavaBean是两个常见的关键字，很容易混淆。

POJO的全名是普通普通Java对象/纯旧Java对象。在中文中，它可以翻译为：普通的Java类，带有一些getter/setter方法的类可以称为POJO，但是JavaBean要比POJO复杂得多。

Java bean是一个可重用组件。javabean没有严格的规范。理论上，任何Java类都可以是bean。但是通常，由于javabean是由容器（比如Tomcat）创建的，javabean应该有一个无参数的构造函数。另外，javabean通常需要实现可串行化的接口来实现bean持久化。不能跨进程访问javabean。JavaBean是一种组件技术。

什么是POJO？

POJO是普通旧Java对象的缩写，即JavaBean。VO是viewobject的缩写，它是用于页面显示的JavaBean。VO是POJO。它只用于携带途中的页面显示信息。

java的反射到底是有什么用处？怎么用？

Java反射意味着在运行状态下，对于任何类，我们都可以知道该类具有哪些方法和属性。对于任何对象，我们都可以调用它的方法和属性。我们将此函数称为动态获取对象信息和调用对象方法的反射机制。

[启动类加了@MapperScan("com.jg.pochi.mapper")](mailto:启动类加了@MapperScan(%22com.jg.pochi.mapper%22))其实就不用在mapper接口类加上@Component注解，但IDEA会提示*Could not autowire. No beans of 'SysLogMapper' type found，*因此一般会习惯加上

线程上下文与*ThreadLocal*

\* 线程上下文,ThreadLocal是JDK提供的,如果创建一个ThreadLocal变量

\* 那么每个访问这个变量的线程都会有这个变量的副本,操作的是本地的内存

\* 从而规避了线程不安全的问题

\* 注意,线程用完以后一定要remove本地内存中的本地变量

**JoinPoint 对象**

JoinPoint对象封装了SpringAop中切面方法的信息,在切面方法中添加JoinPoint参数,就可以获取到封装了该方法信息的JoinPoint对象.

**Proceedingjoinpoint** 继承了 JoinPoint。是在JoinPoint的基础上暴露出 proceed 这个方法。proceed很重要，这个是aop代理链执行的方法。

*环绕通知=前置+目标方法执行+后置通知，proceed方法就是用于启动目标方法执行的*

Spring的AOP只能支持到方法级别的切入。换句话说，切入点只能是某个方法。

**环绕通知：**

使用@Around注释，功能最强，却不常用，环绕通知需要携带ProceedingJoinPoint类型的参数。环绕通知类似于动态代理全过程，ProceedingJoinPoint类型参数可以决定是否执行目标方法，且环绕通知必须要有返回值，返回值即为目标方法的返回值，

**面向切面编程AOP**

面向切面编程（AOP是Aspect Oriented Program的首字母缩写）,我们知道，面向对象的特点是**继承**、**多态**和**封装**。而封装就要求将功能分散到不同的对象中去，这在软件设计中往往称为职责分配。实际上也就是说，让不同的类设计不同的方法。这样代码就分散到一个个的类中去了。这样做的好处是降低了代码的复杂程度，使类可重用。

但是人们也发现，在分散代码的同时，也增加了代码的重复性。什么意思呢？比如说，我们在两个类中，可能都需要在每个方法中做日志。按面向对象的设计方法，我们就必须在两个类的方法中都加入日志的内容。也许他们是完全相同的，但就是因为面向对象的设计让类与类之间无法联系，而不能将这些重复的代码统一起来。

也许有人会说，那好办啊，我们可以将这段代码写在一个独立的类独立的方法里，然后再在这两个类中调用。但是，这样一来，这两个类跟我们上面提到的独立的类就有耦合了，它的改变会影响这两个类。那么，有没有什么办法，能让我们在需要的时候，随意地加入代码呢？这种在运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面的编程。

一般而言，我们管**切入到指定类指定方法的代码片段**称为切面，而切入到**哪些类、哪些方法**则叫切入点。有了AOP，我们就可以把几个类共有的代码，抽取到一个切片中，等到需要时再切入对象中去，从而改变其原有的行为。这样看来，AOP其实只是OOP的补充而已。OOP从横向上区分出一个个的类来，而AOP则从纵向上向对象中加入特定的代码。有了AOP，OOP变得立体了。如果加上时间维度，AOP使OOP由原来的二维变为三维了，由平面变成立体了。从技术上来说，AOP基本上是通过代理机制实现的。

AOP在编程历史上可以说是里程碑式的，对OOP编程是一种十分有益的补充。

AOP不一定都像Spring AOP那样，是在运行时生成代理对象来织入的，还可以在编译期、类加载期织入，比如AspectJ。

*Spring bean默认作用域单例。*

Throwable

1. Throwable是Error和Exception的父类，用来定义所有可以作为异常抛出来的类。

Error和Exception区分：

Error是编译时错误和系统错误，系统错误在除特殊情况下，都不需要你来关心，基本不会出现。而编译时错误，如果你使用了编译器，那么编译器会提示。

Exception则是可以被抛出的基本类型，我们需要主要关心的也是这个类。

Exception又分为RunTimeException和其他Exception。

RunTimeException和其他Exception区分：

其他Exception，受检查异常。可以理解为错误，必须要开发者解决以后才能编译通过，解决的方法有两种，1：throw到上层，2，try-catch处理。

RunTimeException：运行时异常，又称不受检查异常，不受检查！不受检查！！不受检查！！！重要的事情说三遍，因为不受检查，所以在代码中可能会有RunTimeException时Java编译检查时不会告诉你有这个异常，但是在实际运行代码时则会暴露出来，比如经典的1/0，空指针等。如果不处理也会被Java自己处理。

**实体类**

package com.jg.pochi.pojo;  
​  
import lombok.Data;  
​  
/\*\*  
\* @Author: 杨德石  
\* @Date: 2020/9/21 22:59  
\* @Version 1.0  
\*/  
@Data  
public class SysLog {  
​  
   /\*\*  
    \* ID  
    \*/  
   private Long logId;  
​  
   /\*\*  
    \* 请求路径  
    \*/  
   private String logUrl;  
​  
   /\*\*  
    \* 参数  
    \*/  
   private String logParams;  
​  
   /\*\*  
    \* 状态，1正常，0异常  
    \*/  
   private Integer logStatus;  
​  
   /\*\*  
    \* 异常文本  
    \*/  
   private String logMessage;  
​  
   /\*\*  
    \* 浏览器UA标识  
    \*/  
   private String logUa;  
​  
   /\*\*  
    \* controller  
    \*/  
   private String logController;  
​  
   /\*\*  
    \* 请求方式，GET、POST等等  
    \*/  
   private String logMethod;  
​  
   /\*\*  
    \* 响应时间  
    \*/  
   private Long logTime;  
​  
   /\*\*  
    \* 返回值  
    \*/  
   private String logResult;  
​  
   /\*\*  
    \* 请求ip  
    \*/  
   private String logIp;  
​  
   /\*\*  
    \* 创建时间  
    \*/  
   private String createdDate;  
​  
   /\*\*  
    \* 创建人账号  
    \*/  
   private String createdBy;  
​  
}  
​

**线程上下文**

新建包 context ，包下新建类 SystemContext

该类的原理涉及到JMM，暂时不讲解，目前只需要理解成是给每个线程提供一个全局变量即可。

package com.jg.pochi.context;  
  
import com.jg.pochi.pojo.SysLog;  
import lombok.Data;  
  
/\*\*  
 \* 本地线程上下文  
 \* 用来存储在同一个线程中可能会用到的全局变量  
 \*  
 \* @author 杨德石  
 \*/  
@Data  
public class SystemContext {  
  
 /\*\*  
 \* 日志实体  
 \*/  
 private SysLog logger = new SysLog();  
  
 /\*\*  
 \* 是否记录日志  
 \*/  
 private boolean isLog = false;  
  
 /\*\*  
 \* 线程本地内存中的变量  
 \*/  
 private static ThreadLocal<SystemContext> threadLocal = new ThreadLocal<>();  
  
 public static SystemContext get() {  
 if (threadLocal.get() == null) {  
 SystemContext threadLocalContext = new SystemContext();  
 threadLocal.set(threadLocalContext);  
 }  
 return threadLocal.get();  
 }  
  
 public void remove() {  
 threadLocal.remove();  
 }  
}

**mapper**

package com.jg.pochi.mapper;  
  
import com.jg.pochi.pojo.SysLog;  
import org.springframework.stereotype.Component;  
  
/\*\*  
 \* @Author: 杨德石  
 \* @Date: 2020/9/21 23:06  
 \* @Version 1.0  
 \*/  
@Component  
public interface SysLogMapper {  
  
 /\*\*  
 \* 保存日志  
 \* @param logger  
 \*/  
 void save(SysLog logger);  
}

**xml**

<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
<mapper namespace="com.jg.pochi.mapper.SysLogMapper">  
 <!-- 通用查询映射结果 -->  
 <resultMap id="BaseResultMap" type="com.jg.pochi.pojo.SysLog">  
 <id column="log\_id" property="logId"/>  
 <result column="log\_url" property="logUrl"/>  
 <result column="log\_params" property="logParams"/>  
 <result column="log\_status" property="logStatus"/>  
 <result column="log\_message" property="logMessage"/>  
 <result column="log\_ua" property="logUa"/>  
 <result column="log\_controller" property="logController"/>  
 <result column="log\_method" property="logMethod"/>  
 <result column="log\_time" property="logTime"/>  
 <result column="log\_result" property="logResult"/>  
 <result column="log\_ip" property="logIp"/>  
 <result column="created\_date" property="createdDate"/>  
 <result column="created\_by" property="createdBy"/>  
 </resultMap>  
  
 <!-- 通用查询结果列 -->  
 <sql id="Base\_Column\_List">  
 log\_id, log\_url, log\_params, log\_status, log\_message, log\_ua, log\_controller, log\_method, log\_time, log\_result, log\_ip, created\_date, created\_by  
 </sql>  
 <insert id="save">  
 insert into mg\_sys\_log (log\_id, log\_url, log\_params, log\_status, log\_message, log\_ua, log\_controller,  
 log\_method, log\_time, log\_result, log\_ip, created\_by)  
 values (#{logId}, #{logUrl}, #{logParams}, #{logStatus}, #{logMessage}, #{logUa}, #{logController},  
 #{logMethod}, #{logTime}, #{logResult},  
 #{logIp}, #{createdBy})  
 </insert>  
  
</mapper>

**service**

package com.jg.pochi.service;  
​  
import com.jg.pochi.pojo.SysLog;  
​  
/\*\*  
\* @Author: 杨德石  
\* @Date: 2020/9/21 23:08  
\* @Version 1.0  
\*/  
public interface SysLogService {  
​  
   /\*\*  
    \* 保存日志  
    \* @param logger  
    \*/  
   void save(SysLog logger);  
}  
​

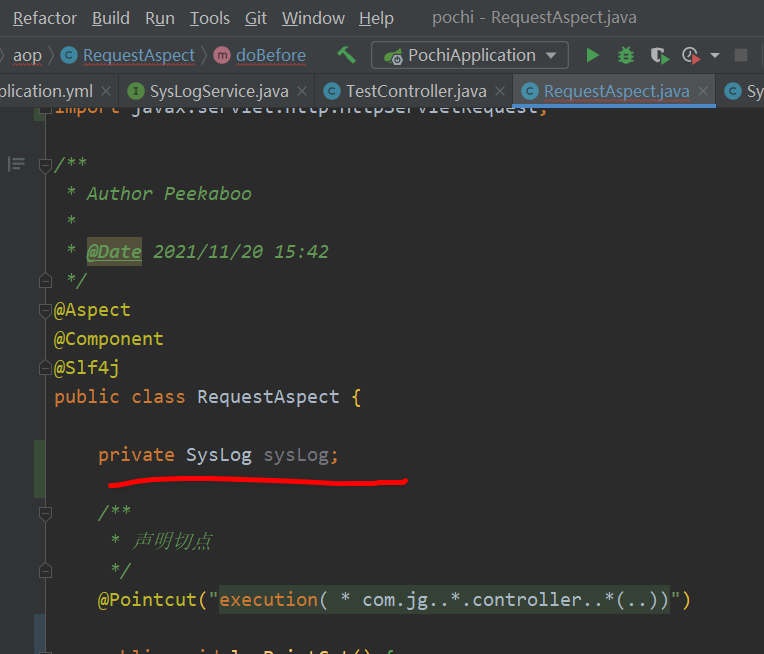
package com.jg.pochi.service.impl;  
​  
import com.jg.pochi.mapper.SysLogMapper;  
import com.jg.pochi.pojo.SysLog;  
import com.jg.pochi.service.SysLogService;  
import com.jg.pochi.utils.IdWorker;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
​  
/\*\*  
\* @Author: 杨德石  
\* @Date: 2020/9/21 23:09  
\* @Version 1.0  
\*/  
@Service  
public class SysLogServiceImpl implements SysLogService {  
​  
   @Autowired  
   private SysLogMapper sysLogMapper;  
   @Autowired  
   private IdWorker idWorker;  
​  
   @Override  
   public void save(SysLog logger) {  
       logger.setLogId(idWorker.nextId());  
       sysLogMapper.save(logger);  
  }  
}  
​

**TestController**

package com.jg.pochi.controller;  
​  
import com.jg.pochi.utils.Result;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RequestMethod;  
import org.springframework.web.bind.annotation.RestController;  
​  
/\*\*  
\* @Author: 杨德石  
\* @Date: 2020/9/21 23:21  
\* @Version 1.0  
\*/  
@RestController  
public class TestController {  
​  
   @RequestMapping(value = "/test", method = RequestMethod.GET)  
   public Result<?> test() {  
       return new Result<>("操作成功");  
  }  
​  
}  
​

**RequestAspect**

遇到问题：如果是这样直接注入的话，遇到2个请求同时近来的情况，就



切面类，用于记录日志

package com.jg.pochi.aop;  
  
import com.alibaba.fastjson.JSON;  
import com.jg.pochi.content.SystemContext;  
import com.jg.pochi.enums.StateEnums;  
import com.jg.pochi.pojo.SysLog;  
import com.jg.pochi.service.SysLogService;  
import com.jg.pochi.utils.StringUtils;  
import lombok.extern.slf4j.Slf4j;  
import org.aspectj.lang.JoinPoint;  
import org.aspectj.lang.ProceedingJoinPoint;  
import org.aspectj.lang.annotation.\*;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Component;  
import org.springframework.web.context.request.RequestContextHolder;  
import org.springframework.web.context.request.ServletRequestAttributes;  
import org.springframework.web.multipart.MultipartFile;  
  
import javax.servlet.ServletRequest;  
import javax.servlet.http.HttpServletRequest;  
  
*/\*\*  
 \* Author Peekaboo  
 \*  
 \* Date 2021/11/20 15:42  
 \*/*@Aspect  
@Component  
@Slf4j  
public class RequestAspect *{* @Autowired  
 private SysLogService sysLogService;  
  
 */\*\*  
 \* 声明切点  
 \*/* @Pointcut*(*"execution( \* com.jg..\*.controller..\*(..))"*)* public void logPointCut*() {  
  
 }  
  
 /\*\*  
 \* 前置通知  
 \*  
 \* throws Exception  
 \*/* @Before*(*"logPointCut()"*)* public void doBefore*(*JoinPoint joinPoint*)* throws Exception *{* // 接收到请求，记录请求内容  
 ServletRequestAttributes attributes = *(*ServletRequestAttributes*)* RequestContextHolder.*getRequestAttributes()*;  
 assert attributes != null;  
 // 获取request  
 HttpServletRequest request = attributes.getRequest*()*;  
 // 获取请求地址  
 String uri = request.getRequestURI*()*;  
 // 记录日志  
 // 日志输出基本信息  
 *log*.info*(*"请求地址：{}", uri*)*;  
 *log*.info*(*"请求方式：{}", request.getMethod*())*;  
 // 获取请求IP  
 String remoteIp = StringUtils.*getRemoteIp(*request*)*;  
 *log*.info*(*"IP：{}", remoteIp*)*;  
 // 获取请求的controller  
 String controllerName = joinPoint.getSignature*()*.getDeclaringTypeName*()*;  
 *log*.info*(*"方法：{}.{}", controllerName, joinPoint.getSignature*()*.getName*())*;  
 // 记录参数  
 Object*[]* args = joinPoint.getArgs*()*;  
 // 记录日志条件：参数不为空，并且第一个参数不是request也不是MultipartFile  
 boolean logParamFlag = args != null && args.length > 0 && !*(*args*[*0*]* instanceof ServletRequest*)* && !*(*args*[*0*]* instanceof MultipartFile*)*;  
 SysLog sysLog = SystemContext.*get()*.getSysLog*()*;  
 if *(*logParamFlag*) {* String param = JSON.*toJSONString(*args*[*0*])*;  
 *log*.info*(*"请求参数：{}", param*)*;  
 sysLog.setLogParams*(*param*)*;  
 *}* // 记录日志  
 sysLog.setLogUrl*(*uri*)*;  
 sysLog.setLogStatus*(*StateEnums.*REQUEST\_SUCCESS*.getCode*())*;  
 sysLog.setLogMethod*(*request.getMethod*())*;  
 sysLog.setLogIp*(*remoteIp*)*;  
 sysLog.setLogUa*(*request.getHeader*(*"user-Agent"*))*;  
 sysLog.setLogController*(*controllerName*)*;  
 *}  
  
 /\*\*  
 \** ***@param*** *pjp  
 \* proceed很重要，这个是aop代理链执行的方法。  
 \* Around 修饰的环绕通知类型，是将整个目标方法封装起来  
 \* throws Throwable  
 \*/* @Around*(*"logPointCut()"*)* public Object doAround*(*ProceedingJoinPoint pjp*)* throws Throwable*{* //记录方法,执行时间  
 long startTime = System.*currentTimeMillis()*;  
 Object ob = pjp.proceed*()*;  
 long time = System.*currentTimeMillis()* - startTime;  
 *log*.info*(*"方法执行耗时:{}",time*)*;  
 SysLog sysLog = SystemContext.*get()*.getSysLog*()*;  
 sysLog.setLogTime*(*time*)*;  
 return ob;  
 *}  
  
 /\*\*  
 \* 指定参数名(doAround返回值)和切点  
 \** ***@param*** *ret  
 \*/* @AfterReturning*(*returning = "ret",pointcut = "logPointCut()"*)* public void doAfter*(*Object ret*){* String result = JSON.*toJSONString(*ret*)*;  
 *log*.info*(*"返回值{}" , result*)*;  
 SysLog sysLog = SystemContext.*get()*.getSysLog*()*;  
 sysLog.setLogResult*(*result*)*;  
 sysLogService.save*(*sysLog*)*;  
 SystemContext.*remove()*;  
 *}  
  
 /\*\*  
 \* 异常通知,发生异常走这里  
 \** ***@param*** *joinPoint  
 \** ***@param*** *throwable  
 \*/* @AfterThrowing*(*pointcut = "logPointCut()", throwing = "throwable"*)* public void doException*(*JoinPoint joinPoint, Throwable throwable*){* SysLog sysLog = SystemContext.*get()*.getSysLog*()*;  
 sysLog.setLogStatus*(*StateEnums.*NOT\_ENABLE*.getCode*())*;  
 sysLog.setLogMessage*(*StateEnums.*NOT\_ENABLE*.getMsg*())*;  
 sysLog.setLogTime*(*0L*)*;  
 sysLogService.save*(*sysLog*)*;  
 SystemContext.*remove()*;  
 *}  
}*

​

访问测试接口，观察日志记录情况

**3. 前端搭建**

**3.1 拉取项目**

github或者gitee拉取 vue-element-admin

git clone https://github.com/PanJiaChen/vue-element-admin.git

**3.2 装包**

安装 nrm 包管理工具

npm i nrm -g

查看所有镜像

nrm ls

切换淘宝镜像

nrm use taobao

安装 node-sass，该包在github上，使用nrm修改镜像下载也很慢，需要手动设置镜像下载

npm i node-sass --sass\_binary\_site=https://npm.taobao.org/mirrors/node-sass/

最后，安装剩下所有的包

npm i

**3.3 启动**

npm run dev

**3.4 包结构介绍**

├── build                      # 构建相关  
├── mock                       # 项目mock 模拟数据  
├── plop-templates             # 基本模板  
├── public                     # 静态资源  
│   │── favicon.ico            # favicon图标  
│   └── index.html             # html模板  
├── src                        # 源代码  
│   ├── api                    # 所有请求  
│   ├── assets                 # 主题 字体等静态资源  
│   ├── components             # 全局公用组件  
│   ├── directive              # 全局指令  
│   ├── filters                # 全局 filter  
│   ├── icons                  # 项目所有 svg icons  
│   ├── lang                   # 国际化 language  
│   ├── layout                 # 全局 layout  
│   ├── router                 # 路由  
│   ├── store                  # 全局 store管理  
│   ├── styles                 # 全局样式  
│   ├── utils                  # 全局公用方法  
│   ├── vendor                 # 公用vendor  
│   ├── views                  # views 所有页面  
│   ├── App.vue                # 入口页面  
│   ├── main.js                # 入口文件 加载组件 初始化等  
│   └── permission.js          # 权限管理  
├── tests                      # 测试  
├── .env.xxx                   # 环境变量配置  
├── .eslintrc.js               # eslint 配置项  
├── .babelrc                   # babel-loader 配置  
├── .travis.yml                # 自动化CI配置  
├── vue.config.js              # vue-cli 配置  
├── postcss.config.js          # postcss 配置  
└── package.json               # package.json

**3.5 内容调整**

1. 打开 src->main.js，找到第31行，删除以下代码。这里是mock的地址，会影响我们开发

if (process.env.NODE\_ENV === 'production') {  
 const { mockXHR } = require('../mock')  
 mockXHR()  
}

1. 打开 src->router->index.js，将 constantRoutes 中的菜单路由放到 asyncRoutes 下，这里是动态路由，我们把菜单都放到动态路由这里，便于我们后面动态设置菜单。
2. 删除 plop-template 文件夹。这里是页面模板，但是模板比较简单，对开发帮助不大
3. 删除 mock 文件夹，并找到 vue.config.js 文件，删除 devServer 下和 mock 相关的内容，在下面加上以下内容。这里是配置代理，目的是从前端解决跨域问题。删除后重启可能会崩溃，重启 VsCode即可

proxy: {  
  [process.env.VUE\_APP\_BASE\_API]: {  
       target: `http://127.0.0.1:8080`,  
       changeOrigin: true,  
       pathRewrite: {  
          ['^' + process.env.VUE\_APP\_BASE\_API]: ''  
      }  
  }  
}

1. 修改 .env.xxx 文件，所有的 VUE\_APP\_BASE\_API 都改成 /api
2. 进入 request.js 中，修改为以下内容

import axios from 'axios'  
import { Message } from 'element-ui'  
import store from '@/store'  
import { getToken } from '@/utils/auth'  
  
// create an axios instance  
const service = axios.create({  
 baseURL: process.env.VUE\_APP\_BASE\_API, // url = base url + request url  
 // withCredentials: true, // send cookies when cross-domain requests  
 timeout: 5000 // request timeout  
})  
  
// request interceptor  
service.interceptors.request.use(  
 config => {  
 // do something before request is sent  
  
 if (store.getters.token) {  
 // let each request carry token  
 // ['X-Token'] is a custom headers key  
 // please modify it according to the actual situation  
 config.headers['Authorization'] = getToken()  
 }  
 return config  
 },  
 error => {  
 // do something with request error  
 console.log(error) // for debug  
 return Promise.reject(error)  
 }  
)  
  
// response interceptor  
service.interceptors.response.use(  
 response => {  
 const res = response.data  
  
 // if the custom code is not 20000, it is judged as an error.  
 if (res.code !== 20000) {  
 Message({  
 message: res.msg || 'Error',  
 type: 'error',  
 duration: 5 \* 1000  
 })  
  
 return Promise.reject(new Error(res.message || 'Error'))  
 } else {  
 return res  
 }  
 },  
 error => {  
 console.log('err' + error) // for debug  
 Message({  
 message: '操作失败',  
 type: 'error',  
 duration: 5 \* 1000  
 })  
 return Promise.reject(error)  
 }  
)  
  
export default service