# 日志管理设计说明

## 业务设计说明

本模块主要是实现对用户行为日志(例如谁在什么时间点执行了什么操作,访问了哪些方法,传递的什么参数,执行时长等)进行记录、查询、删除等操作。其表设计语句如下：

**CREATE** **TABLE** `sys\_logs` (

`id` bigint(20) **NOT** **NULL** AUTO\_INCREMENT,

`username` **varchar**(50) **DEFAULT** **NULL** COMMENT '用户名',

`operation` **varchar**(50) **DEFAULT** **NULL** COMMENT '用户操作',

`method` **varchar**(200) **DEFAULT** **NULL** COMMENT '请求方法',

`params` **varchar**(5000) **DEFAULT** **NULL** COMMENT '请求参数',

`**time**` bigint(20) **NOT** **NULL** COMMENT '执行时长(毫秒)',

`ip` **varchar**(64) **DEFAULT** **NULL** COMMENT 'IP地址',

`createdTime` datetime **DEFAULT** **NULL** COMMENT '创建时间',

**PRIMARY** **KEY** (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=1 **DEFAULT** CHARSET=utf8 COMMENT='系统日志';

## 原型设计说明

基于用户需求，实现静态页面(html/css/js)，通过静态页面为用户呈现基本需求实现，如图-1所示。

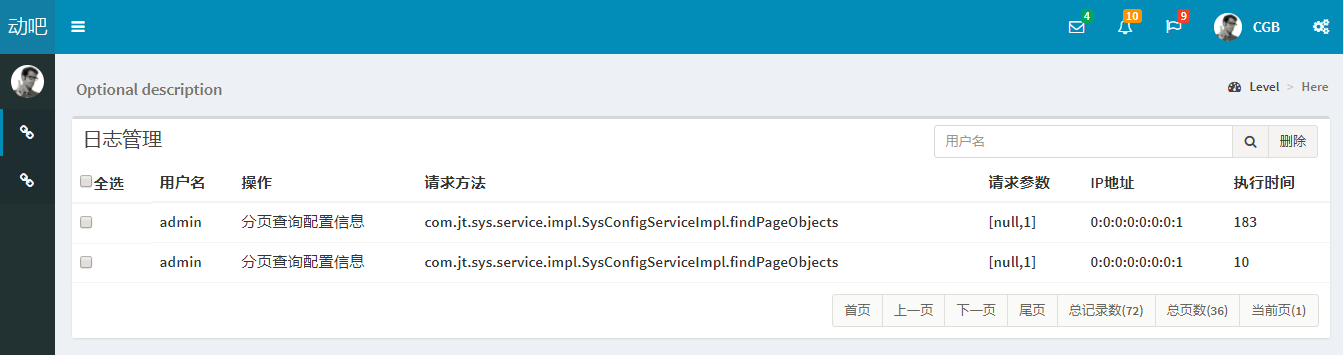


图-1

说明:假如客户对此原型进行了确认,后续则可以基于此原型进行研发。

## API设计说明

日志业务后台API分层架构及调用关系如图-2所示：

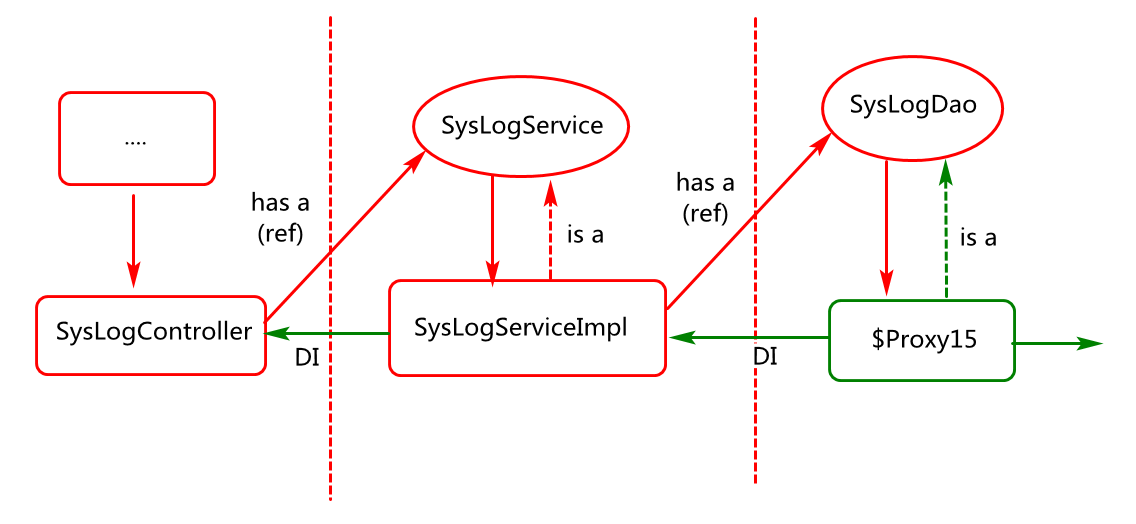


图-2

说明：分层目的主要将复杂问题简单化，实现各司其职，各尽所能。

# 日志管理列表页面呈现

## 服务端实现

### Controller实现

* 业务描述与设计实现

基于日志菜单管理的请求业务，在PageController中添加doLogUI方法，doPageUI方法分别用于返回日志列表页面，日志分页页面。

* 关键代码设计与实现

第一步：在PageController中定义返回日志列表的方法。代码如下：

@RequestMapping("log/log\_list")

**public** String doLogUI() {

**return** "sys/log\_list";

}

第二步：在PageController中定义用于返回分页页面的方法。代码如下：

@RequestMapping("doPageUI")

**public** String doPageUI() {

**return** "common/page";

}

## 客户端实现

### 日志菜单事件处理

* 业务描述与设计

首先准备日志列表页面(/templates/pages/sys/log\_list.html)，然后在starter.html页面中点击日志管理菜单时异步加载日志列表页面。

* 关键代码设计与实现

找到项目中的starter.html 页面，页面加载完成以后，注册日志管理菜单项的点击事件，当点击菜单管理时，执行事件处理函数。关键代码如下：

$(function(){

doLoadUI("load-log-id","log/log\_list")

})

function doLoadUI(id,url){

$("#"+id).click(function(){

$("#mainContentId").load(url);

});

}

### 日志列表页面事件处理

* 业务描述与设计实现

当日志列表页面加载完成以后异步加载分页页面（page.html）。

* 关键代码设计与实现：

在log\_list.html页面中异步加载page页面，这样可以实现分页页面重用，哪里需要分页页面，哪里就进行页面加载即可。关键代码如下：

$(function(){

$("#pageId").load("doPageUI");

});

说明：数据加载通常是一个相对比较耗时操作，为了改善用户体验，可以先为用户呈现一个页面，数据加载时，显示数据正在加载中,数据加载完成以后再呈现数据。这样也可满足现阶段不同类型客户端需求(例如手机端,电脑端,电视端,手表端。)

# 日志管理列表数据呈现

## 数据架构分析

日志查询服务端数据基本架构，如图-3所示。

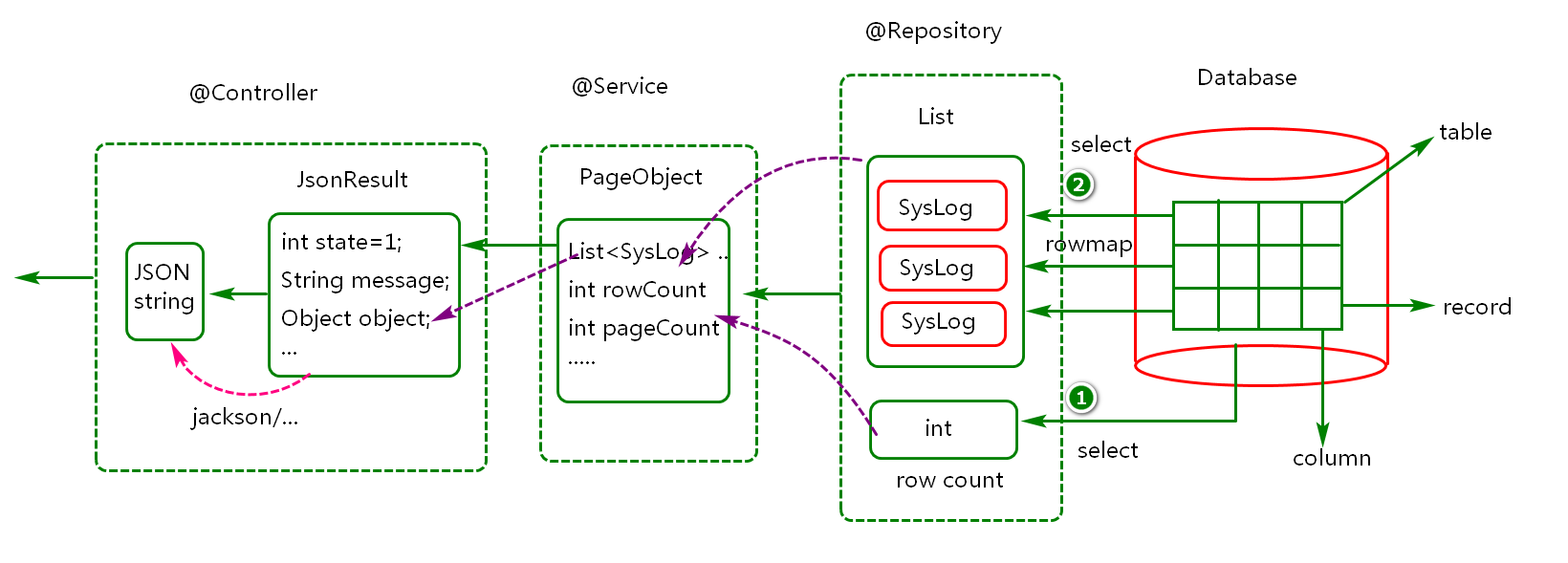


图-3

## 服务端API架构及业务时序图分析

服务端日志分页查询代码基本架构，如图-4所示：

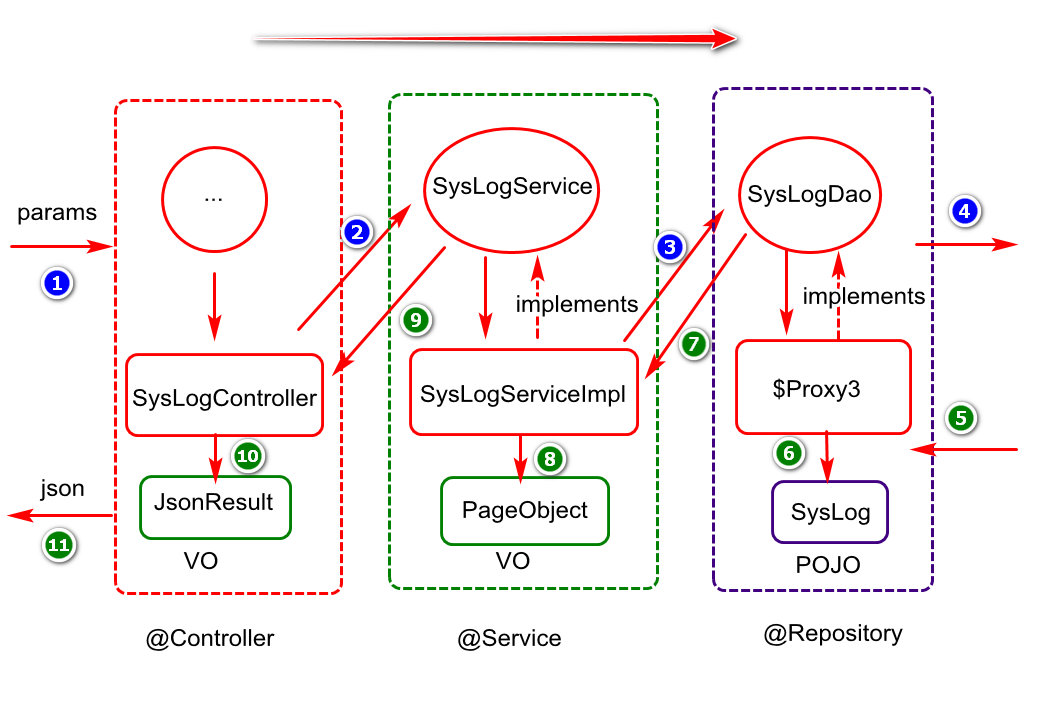


图-4

服务端日志列表数据查询时序图，如图-5所示：

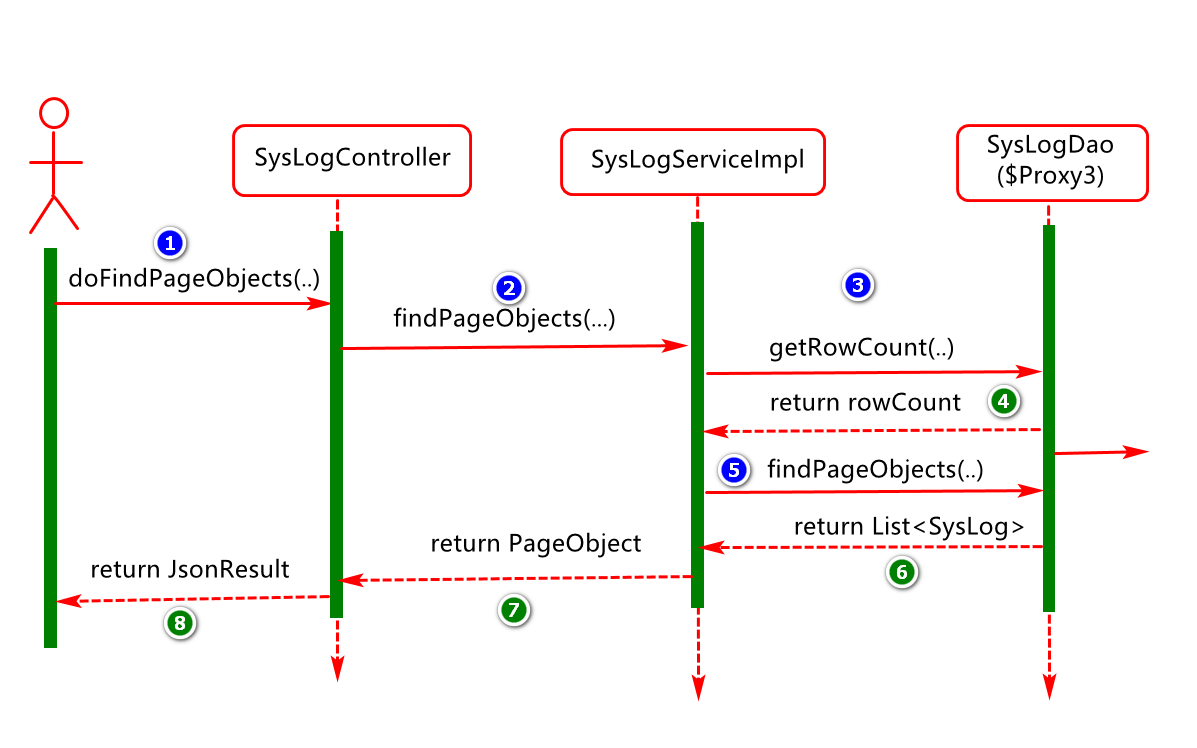


图-5

## 服务端关键业务及代码实现

### Entity类实现

* 业务描述及设计实现

构建实体对象（POJO）封装从数据库查询到的记录，一行记录映射为内存中一个的这样的对象。对象属性定义时尽量与表中字段有一定的映射关系，并添加对应的set/get/toString等方法，便于对数据进行更好的操作。

* 关键代码分析及实现

**package** com.cy.pj.sys.entity;

**import** java.io.Serializable;

**import** java.util.Date;

**public** **class** SysLog **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** Integer id;

//用户名

**private** String username;

//用户操作

**private** String operation;

//请求方法

**private** String method;

//请求参数

**private** String params;

//执行时长(毫秒)

**private** Long time;

//IP地址

**private** String ip;

//创建时间

**private** Date createdTime;

/\*\*设置：\*/

**public** **void** setId(Integer id) {

**this**.id = id;

}

/\*\*获取：\*/

**public** Integer getId() {

**return** id;

}

/\*\*设置：用户名\*/

**public** **void** setUsername(String username) {

**this**.username = username;

}

/\*\* 获取：用户名\*/

**public** String getUsername() {

**return** username;

}

/\*\*设置：用户操作\*/

**public** **void** setOperation(String operation) {

**this**.operation = operation;

}

/\*\*获取：用户操作\*/

**public** String getOperation() {

**return** operation;

}

/\*\*设置：请求方法\*/

**public** **void** setMethod(String method) {

**this**.method = method;

}

/\*\*获取：请求方法\*/

**public** String getMethod() {

**return** method;

}

/\*\* 设置：请求参数\*/

**public** **void** setParams(String params) {

**this**.params = params;

}

/\*\* 获取：请求参数 \*/

**public** String getParams() {

**return** params;

}

/\*\*设置：IP地址 \*/

**public** **void** setIp(String ip) {

**this**.ip = ip;

}

/\*\* 获取：IP地址\*/

**public** String getIp() {

**return** ip;

}

/\*\* 设置：创建时间\*/

**public** **void** setCreateDate(Date createdTime) {

**this**.createdTime = createdTime;

}

/\*\* 获取：创建时间\*/

**public** Date getCreatedTime() {

**return** createdTime;

}

**public** Long getTime() {

**return** time;

}

**public** **void** setTime(Long time) {

**this**.time = time;

}

}

说明：通过此对象除了可以封装从数据库查询的数据，还可以封装客户端请求数据，实现层与层之间数据的传递。

### Dao接口实现

* 业务描述及设计实现

通过数据层对象，基于业务层参数数据查询日志记录总数以及当前页要呈现的用户行为日志信息。

* 关键代码分析及实现：

第一步：定义数据层接口对象，通过将此对象保证给业务层以提供日志数据访问。代码如下：

@Mapper

public interface SysLogDao {

}

第二步：在SysLogDao接口中添加getRowCount方法用于按条件统计记录总数。代码如下：

/\*\*

\* 基于条件查询总记录数

\* @param username 查询条件(例如查询哪个用户的日志信息)

\* @return 总记录数(基于这个结果可以计算总页数)

\* 说明：假如如下方法没有使用注解修饰，在基于名字进行查询

\* 时候会出现There is no getter for property named

\* 'username' in 'class java.lang.String'

\*/

int getRowCount(@Param("username") String username);

}

第三步：在SysLogDao接口中添加findPageObjects方法，基于此方法实现当前页记录的数据查询操作。代码如下：

/\*\*

\* 基于条件分页查询日志信息

\* @param username 查询条件(例如查询哪个用户的日志信息)

\* @param startIndex 当前页的起始位置

\* @param pageSize 当前页的页面大小

\* @return 当前页的日志记录信息

\* 数据库中每条日志信息封装到一个SysLog对象中

\*/

List<SysLog> findPageObjects(

@Param("username")String username,

@Param("startIndex")Integer startIndex,

@Param("pageSize")Integer pageSize);

说明：

1. 当DAO中方法参数多余一个时尽量使用@Param注解进行修饰并指定名字，然后再Mapper文件中便可以通过类似#{username}方式进行获取，否则只能通过#{arg0}，#{arg1}或者#{param1}，#{param2}等方式进行获取。
2. 当DAO方法中的参数应用在动态SQL中时无论多少个参数，尽量使用@Param注解进行修饰并定义。

### Mapper文件实现

* 业务描述及设计实现

基于Dao接口创建映射文件，在此文件中通过相关元素(例如select)描述要执行的数据操作。

* 关键代码设计及实现

第一步：在映射文件的设计目录(mapper/sys)中添加SysLogMapper.xml映射文件，代码如下：

<?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.cy.pj.sys.dao.SysLogDao"*>

</mapper>

第二步：在映射文件中添加sql元素实现，SQL中的共性操作，代码如下：

<sql id=*"queryWhereId"*>

<where>

<if test=*"username!=null and username!=''"*>

username like concat("%",#{username},"%")

</if>

</where>

</sql>

第三步：在映射文件中添加id为getRowCount元素，按条件统计记录总数，代码如下：

<select id=*"getRowCount"*

resultType=*"int"*>

select count(\*)

from sys\_Logs

<include refid=*"queryWhereId"*/>

</select>

第四步：在映射文件中添加id为findPageObjects元素，实现分页查询。代码如下：

<select id=*"findPageObjects"*

resultType=*"com.cy.pj.sys.entity.SysLog"*>

select \*

from sys\_Logs

<include refid=*"queryWhereId"*/>

order by createdTime desc

limit #{startIndex},#{pageSize}

</select>

思考:

1. 动态sql:基于用于需求动态拼接SQL
2. Sql标签元素的作用是什么?对sql语句中的共性进行提取,以遍实现更好的复用.
3. Include标签的作用是什么?引入使用sql标签定义的元素

### Service接口及实现类

* 业务描述与设计实现

业务层主要是实现模块中业务逻辑的处理。在日志分页查询中，业务层对象首先要通过业务方法中的参数接收控制层数据(例如username,pageCurrent)并校验。然后基于用户名进行总记录数的查询并校验，再基于起始位置及页面大小进行当前页记录的查询，最后对查询结果进行封装并返回。

* 关键代码设计及实现

业务值对象定义，基于此对象封装数据层返回的数据以及计算的分页信息，具体代码参考如下：

**package** com.cy.pj.common.vo;

**public** **class** PageObject<T> **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 6780580291247550747L;//类泛型

/\*\*当前页的页码值\*/

**private** Integer pageCurrent=1;

/\*\*页面大小\*/

**private** Integer pageSize=3;

/\*\*总行数(通过查询获得)\*/

**private** Integer rowCount=0;

/\*\*总页数(通过计算获得)\*/

**private** Integer pageCount=0;

/\*\*当前页记录\*/

**private** List<T> records;

**public** PageObject(){}

**public** PageObject(Integer pageCurrent, Integer pageSize, Integer rowCount, List<T> records) {

**super**();

**this**.pageCurrent = pageCurrent;

**this**.pageSize = pageSize;

**this**.rowCount = rowCount;

**this**.records = records;

// this.pageCount=rowCount/pageSize;

// if(rowCount%pageSize!=0) {

// pageCount++;

// }

**this**.pageCount=(rowCount-1)/pageSize+1;

}

**public** Integer getPageCurrent() {

**return** pageCurrent;

}

**public** **void** setPageCurrent(Integer pageCurrent) {

**this**.pageCurrent = pageCurrent;

}

**public** Integer getPageSize() {

**return** pageSize;

}

**public** **void** setPageSize(Integer pageSize) {

**this**.pageSize = pageSize;

}

**public** Integer getRowCount() {

**return** rowCount;

}

**public** **void** setRowCount(Integer rowCount) {

**this**.rowCount = rowCount;

}

**public** Integer getPageCount() {

**return** pageCount;

}

**public** **void** setPageCount(Integer pageCount) {

**this**.pageCount = pageCount;

}

**public** List<T> getRecords() {

**return** records;

}

**public** **void** setRecords(List<T> records) {

**this**.records = records;

}

}

定义日志业务接口及方法，暴露外界对日志业务数据的访问，其代码参考如下：

**package** com.cy.pj.sys.service;

**public** **interface** SysLogService {

/\*\*

\* 通过此方法实现分页查询操作

\* **@param** name 基于条件查询时的参数名

\* **@param** pageCurrent 当前的页码值

\* **@return** 当前页记录+分页信息

\*/

PageObject<SysLog> findPageObjects(

String username,

Integer pageCurrent);

}

日志业务接口及实现类，用于具体执行日志业务数据的分页查询操作,其代码参考如下：

**package** com.cy.pj.sys.service.impl;

@Service

**public** **class** SysLogServiceImpl **implements** SysLogService{

@Autowired

**private** SysLogDao sysLogDao;

@Override

**public** PageObject<SysLog> findPageObjects(

String name, Integer pageCurrent) {

//1.验证参数合法性

//1.1验证pageCurrent的合法性，

//不合法抛出IllegalArgumentException异常

**if**(pageCurrent==**null**||pageCurrent<1)

**throw** **new** IllegalArgumentException("当前页码不正确");

//2.基于条件查询总记录数

//2.1) 执行查询

**int** rowCount=sysLogDao.getRowCount(name);

//2.2) 验证查询结果，假如结果为0不再执行如下操作

**if**(rowCount==0)

**throw** **new** ServiceException("系统没有查到对应记录");

//3.基于条件查询当前页记录(pageSize定义为2)

//3.1)定义pageSize

**int** pageSize=2;

//3.2)计算startIndex

**int** startIndex=(pageCurrent-1)\*pageSize;

//3.3)执行当前数据的查询操作

List<SysLog> records=

sysLogDao.findPageObjects(name, startIndex, pageSize);

//4.对分页信息以及当前页记录进行封装

//4.1)构建PageObject对象

PageObject<SysLog> pageObject=**new** PageObject<>();

//4.2)封装数据

pageObject.setPageCurrent(pageCurrent);

pageObject.setPageSize(pageSize);

pageObject.setRowCount(rowCount);

pageObject.setRecords(records);

pageObject.setPageCount((rowCount-1)/pageSize+1);

//5.返回封装结果。

**return** pageObject;

}

}

在当前方法中需要的ServiceException是一个自己定义的异常, 通过自定义异常可更好的实现对业务问题的描述，同时可以更好的提高用户体验。参考代码如下：

**package** com.cy.pj.common.exception;

**public** **class** ServiceException **extends** RuntimeException {

**private** **static** **final** **long** ***serialVersionUID*** = 7793296502722655579L;

**public** ServiceException() {

**super**();

}

**public** ServiceException(String message) {

**super**(message);

// **TODO** Auto-generated constructor stub

}

**public** ServiceException(Throwable cause) {

**super**(cause);

// **TODO** Auto-generated constructor stub

}

}

说明：几乎在所有的框架中都提供了自定义异常，例如MyBatis中的BindingException等。

### Controller类实现

* 业务描述与设计实现

控制层对象主要负责请求和响应数据的处理，例如，本模块首先要通过控制层对象处理请求参数，然后通过业务层对象执行业务逻辑，再通过VO对象封装响应结果(主要对业务层数据添加状态信息)，最后将响应结果转换为JSON格式的字符串响应到客户端。

* 关键代码设计与实现

定义控制层值对象（VO），目的是基于此对象封装控制层响应结果(在此对象中主要是为业务层执行结果添加状态信息)。Spring MVC框架在响应时可以调用相关API（例如jackson）将其对象转换为JSON格式字符串。

**package** com.cy.pj.common.vo;

**public** **class** JsonResult **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = -856924038217431339L;//SysResult/Result/R

/\*\*状态码\*/

**private** **int** state=1;//1表示SUCCESS,0表示ERROR

/\*\*状态信息\*/

**private** String message="ok";

/\*\*正确数据\*/

**private** Object data;

**public** JsonResult() {}

**public** JsonResult(String message){

**this**.message=message;

}

/\*\*一般查询时调用，封装查询结果\*/

**public** JsonResult(Object data) {

**this**.data=data;

}

/\*\*出现异常时时调用\*/

**public** JsonResult(Throwable t){

**this**.state=0;

**this**.message=t.getMessage();

}

**public** **int** getState() {

**return** state;

}

**public** **void** setState(**int** state) {

**this**.state = state;

}

**public** String getMessage() {

**return** message;

}

**public** **void** setMessage(String message) {

**this**.message = message;

}

**public** Object getData() {

**return** data;

}

**public** **void** setData(Object data) {

**this**.data = data;

}

}

定义Controller类，并将此类对象使用Spring框架中的@Controller注解进行标识，表示此类对象要交给Spring管理。然后基于@RequestMapping注解为此类定义根路径映射。代码参考如下：

**package** com.cy.pj.sys.controller;

@Controller

@RequestMapping("/log/")

**public** **class** SysLogController {

}

在Controller类中添加分页请求处理方法，代码参考如下：

@RequestMapping("doFindPageObjects")

@ResponseBody

**public** JsonResult doFindPageObjects(String username,Integer pageCurrent){

PageObject<SysLog> pageObject=

sysLogService.findPageObjects(username,pageCurrent);

**return** **new** JsonResult(pageObject);

}

定义异常处理类，对控制层可能出现的异常，进行统一异常处理，代码参考如下：

**package** com.cy.pj.common.web;

**import** java.util.logging.Logger;

**import** org.springframework.web.bind.annotation.ControllerAdvice;

**import** org.springframework.web.bind.annotation.ExceptionHandler;

**import** org.springframework.web.bind.annotation.ResponseBody;

**import** com.cy.pj.common.vo.JsonResult;

/\*\*全局异常处理类\*/

@ControllerAdvice

**public** **class** GlobalExceptionHandler {

//JDK中的自带的日志API

@ExceptionHandler(RuntimeException.**class**)

@ResponseBody

**public** JsonResult doHandleRuntimeException(

RuntimeException e){

e.printStackTrace();//也可以写日志

**return** **new** JsonResult(e);//封装异常信息

}

}

控制层响应数据处理分析，如图-6所示：

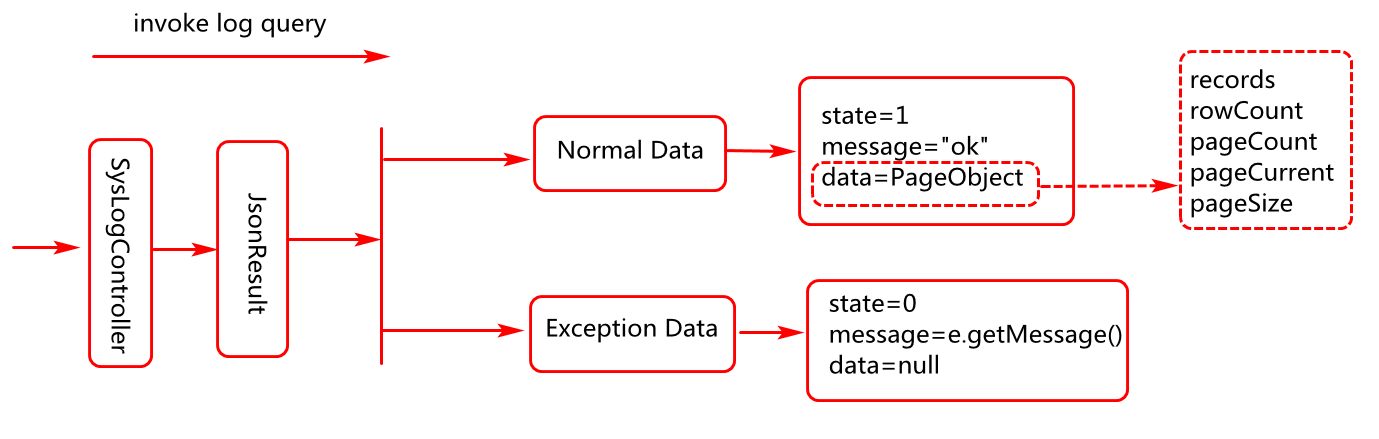


图-6

## 客户端关键业务及代码实现

### 客户端页面事件分析

当用户点击首页日志管理时，其页面流转分析如图-7所示：

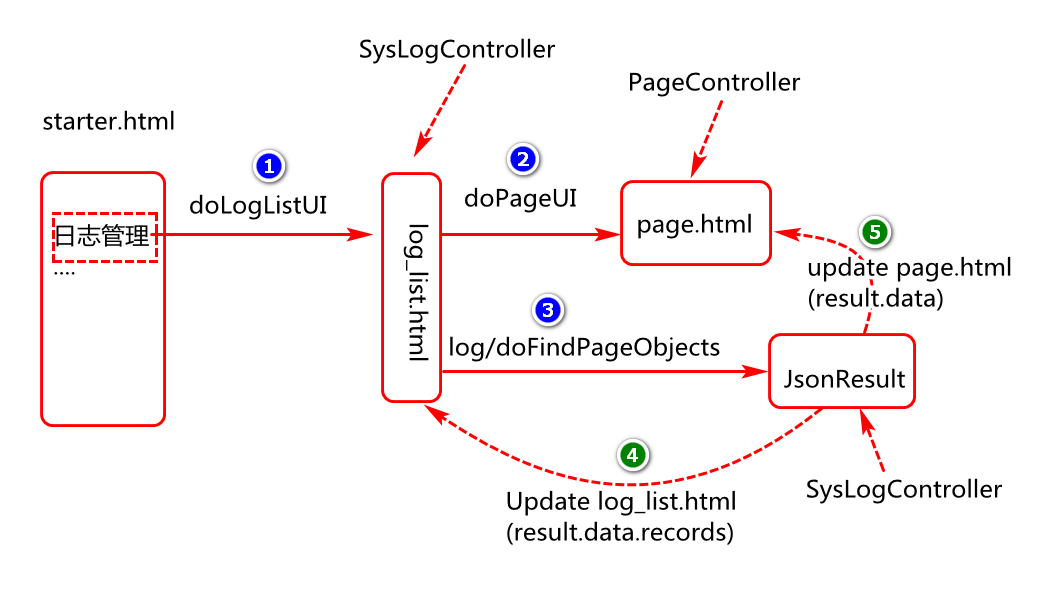


图-7所示：

### 日志列表信息呈现

* 业务描述与设计实现

日志分页页面加载完成以后，向服务端发起异步请求加载日志信息，当日志信息加载完成需要将日志信息、分页信息呈现到列表页面上。

* 关键代码设计与实现

第一步：分页页面加载完成，向服务端发起异步请求，代码参考如下：

$(function(){

//为什么要将doGetObjects函数写到load函数对应的回调内部。

$("#pageId").load("doPageUI",function(){

doGetObjects();

});

}

第二步：定义异步请求处理函数，代码参考如下：

**function** doGetObjects(){

//debugger;//断点调试

//1.定义url和参数

**var** url="log/doFindPageObjects"

**var** params={"pageCurrent":1};//pageCurrent=2

//2.发起异步请求

//请问如下ajax请求的回调函数参数名可以是任意吗？可以,必须符合标识符的规范

$.getJSON(url,params,**function**(result){

//JsonResult->PageObject->List<SysLogs>+...

//请问result是一个字符串还是json格式的js对象？对象

doHandleResponseResult(result);

}

);//特殊的ajax函数

}

result 结果对象分析，如图-8所示：

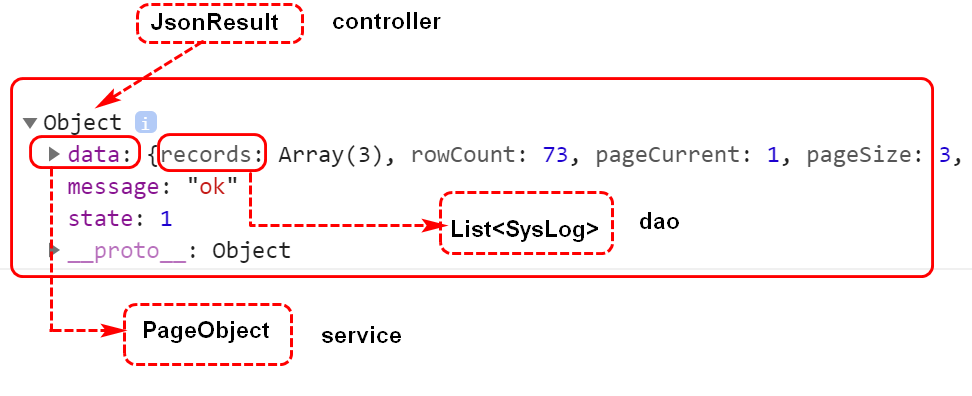


图-8

第三步：定义回调函数，处理服务端的响应结果。代码如下：

**function** doHandleResponseResult (result){ //JsonResult

**if**(result.state==1){//ok

//更新table中tbody内部的数据

doSetTableBodyRows(result.data.records);//将数据呈现在页面上

//更新页面page.html分页数据

//doSetPagination(result.data); //此方法写到page.html中

}**else**{

alert(result.msg);

}

}

第四步：将异步响应结果呈现在table的tbody位置。代码参考如下：

**function** doSetTableBodyRows(records){

//1.获取tbody对象，并清空对象

**var** tBody=$("#tbodyId");

tBody.empty();

//2.迭代records记录，并将其内容追加到tbody

**for**(**var** i **in** records){

//2.1 构建tr对象

**var** tr=$("<tr></tr>");

//2.2 构建tds对象

**var** tds=doCreateTds(records[i]);

//2.3 将tds追加到tr中

tr.append(tds);

//2.4 将tr追加到tbody中

tBody.append(tr);

}

}

第五步：创建每行中的td元素，并填充具体业务数据。代码参考如下：

**function** doCreateTds(data){

**var** tds="<td><input type='checkbox' class='cBox' name='cItem' value='"+data.id+"'></td>"+

"<td>"+data.username+"</td>"+

"<td>"+data.operation+"</td>"+

"<td>"+data.method+"</td>"+

"<td>"+data.params+"</td>"+

"<td>"+data.ip+"</td>"+

"<td>"+data.time+"</td>";

**return** tds;

}

### 分页数据信息呈现

* 业务描述与设计实现

日志信息列表初始化完成以后初始化分页数据（调用setPagination函数），然后再点击上一页，下一页等操作时，更新页码值，执行基于当前页码值的查询。

* 关键代码设计与实现：

第一步：定义doSetPagination方法(实现分页数据初始化)，代码如下：

**function** doSetPagination(page){

//1.始化数据

$(".rowCount").html("总记录数("+page.rowCount+")");

$(".pageCount").html("总页数("+page.pageCount+")");

$(".pageCurrent").html("当前页("+page.pageCurrent+")");

//2.绑定数据(为后续对此数据的使用提供服务)

$("#pageId").data("pageCurrent",page.pageCurrent);

$("#pageId").data("pageCount",page.pageCount);

}

第二步：分页页面page.html中注册点击事件。代码如下：

$(function(){

//事件注册

$("#pageId").on("click",".first,.pre,.next,.last",doJumpToPage);

})

第三步：定义doJumpToPage方法(通过此方法实现当前数据查询)

**function** doJumpToPage(){

//1.获取点击对象的class值

**var** cls=$(**this**).prop("class");//Property

//2.基于点击的对象执行pageCurrent值的修改

//2.1获取pageCurrent,pageCount的当前值

**var** pageCurrent=$("#pageId").data("pageCurrent");

**var** pageCount=$("#pageId").data("pageCount");

//2.2修改pageCurrent的值

**if**(cls=="first"){//首页

pageCurrent=1;

}**else** **if**(cls=="pre"&&pageCurrent>1){//上一页

pageCurrent--;

}**else** **if**(cls=="next"&&pageCurrent<pageCount){//下一页

pageCurrent++;

}**else** **if**(cls=="last"){//最后一页

pageCurrent=pageCount;

}**else**{

return;

}

//3.对pageCurrent值进行重新绑定

$("#pageId").data("pageCurrent",pageCurrent);

//4.基于新的pageCurrent的值进行当前页数据查询

doGetObjects();

}

修改分页查询方法:(看红色框部分)

**function** doGetObjects(){

//debugger;//断点调试

//1.定义url和参数

**var** url="log/doFindPageObjects"

//? 请问data函数的含义是什么？(从指定元素上获取绑定的数据)

//此数据会在何时进行绑定？(setPagination,doQueryObjects)

**var** pageCurrent=$("#pageId").data("pageCurrent");

//为什么要执行如下语句的判定，然后初始化pageCurrent的值为1

//pageCurrent参数在没有赋值的情况下，默认初始值应该为1.

**if**(!pageCurrent) pageCurrent=1;

**var** params={"pageCurrent":pageCurrent};//pageCurrent=2

//2.发起异步请求

//请问如下ajax请求的回调函数参数名可以是任意吗？可以,必须符合标识符的规范

$.getJSON(url,params,**function**(result){

//JsonResult->PageObject->List<SysLogs>+...

//请问result是一个字符串还是json格式的js对象？对象

doHandleResponseResult(result);

}

);//特殊的ajax函数

}

### 列表页面信息查询实现

* 业务描述及设计

当用户点击日志列表的查询按钮时，基于用户输入的用户名进行有条件的分页查询，并将查询结果呈现在页面。

* 关键代码设计与实现：

第一步：日志列表页面加载完成，在查询按钮上进行事件注册。代码如下：

$(".input-group-btn").on("click",".btn-search",doQueryObjects)

第二步：定义查询按钮对应的点击事件处理函数。代码如下：

**function** doQueryObjects(){

//为什么要在此位置初始化pageCurrent的值为1?

//数据查询时页码的初始位置也应该是第一页

$("#pageId").data("pageCurrent",1);

//为什么要调用doGetObjects函数？

//重用js代码，简化jS代码编写。

doGetObjects();

}

第三步：在分页查询函数中追加name参数定义(看红色框部分)，代码如下：

**function** doGetObjects(){

//debugger;//断点调试

//1.定义url和参数

**var** url="log/doFindPageObjects"

//? 请问data函数的含义是什么？(从指定元素上获取绑定的数据)

//此数据会在何时进行绑定？(setPagination,doQueryObjects)

**var** pageCurrent=$("#pageId").data("pageCurrent");

//为什么要执行如下语句的判定，然后初始化pageCurrent的值为1

//pageCurrent参数在没有赋值的情况下，默认初始值应该为1.

**if**(!pageCurrent) pageCurrent=1;

**var** params={"pageCurrent":pageCurrent};

//为什么此位置要获取查询参数的值?

//一种冗余的应用方法，目的时让此函数在查询时可以重用。

**var** username=$("#searchNameId").val();

//如下语句的含义是什么？动态在js对象中添加key/value,

**if**(username) params.username=username;//查询时需要

//2.发起异步请求

//请问如下ajax请求的回调函数参数名可以是任意吗？可以,必须符合标识符的规范

$.getJSON(url,params,**function**(result){

//请问result是一个字符串还是json格式的js对象？对象

doHandleResponseResult(result);

}

);

}

# 日志管理删除操作实现

## 数据架构分析

当用户执行日志删除操作时，客户端与务端交互时的基本数据架构，如图-9所示。

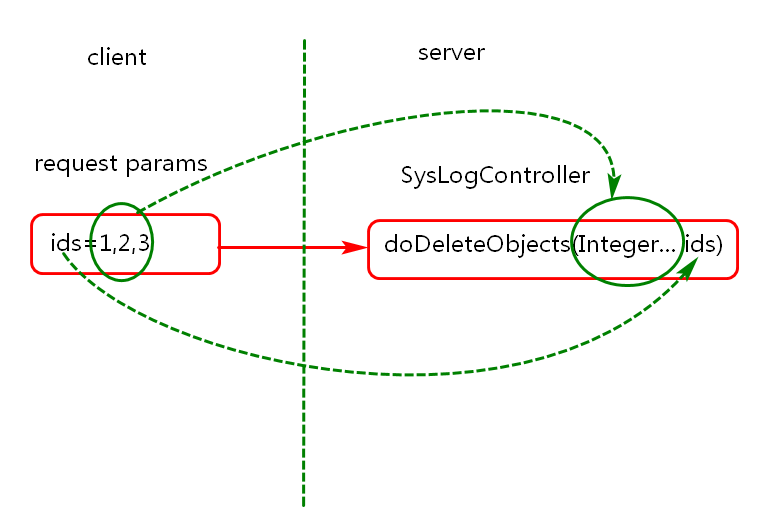


图-9

## 删除业务时序分析

客户端提交删除请求，服务端对象的工作时序分析，如图-10所示。

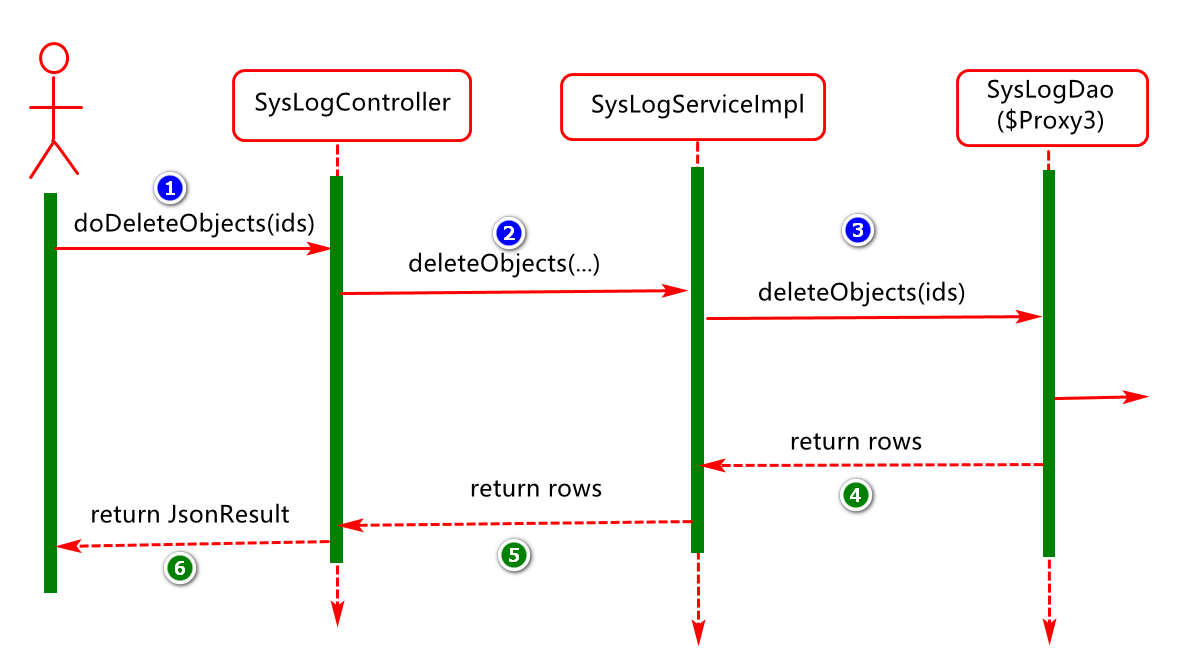


图-10

## 服务端关键业务及代码实现

### Dao接口实现

* 业务描述及设计实现

数据层基于业务层提交的日志记录id，进行日志删除操作。

* 关键代码设计及实现：

在SysLogDao中添加基于id执行日志删除的方法。代码参考如下：

**int** deleteObjects(@Param("ids")Integer… ids);

### Mapper文件实现

* 业务描述及设计实现

在SysLogDao接口对应的映射文件中添加用于执行删除业务的delete元素，此元素内部定义具体的SQL实现。

* 关键代码设计与实现

在SysLogMapper.xml文件添加delete元素，关键代码如下：

<delete id=*"deleteObjects"*>

delete from sys\_Logs

where id in

<foreach collection=*"ids"*

open=*"("*

close=*")"*

separator=*","*

item=*"id"*>

#{id}

</foreach>

</delete>

### Service接口及实现类

* 业务描述与设计实现

在日志业务层定义用于执行删除业务的方法，首先通过方法参数接收控制层传递的多个记录的id，并对参数id进行校验。然后基于日志记录id执行删除业务实现。最后返回业务执行结果。

* 关键代码设计与实现

第一步：在SysLogService接口中，添加基于多个id进行日志删除的方法。关键代码如下：

**int** deleteObjects(Integer… ids) {}

第二步：在SysLogServiceImpl实现类中添加删除业务的具体实现。关键代码如下：

@Override

**public** **int** deleteObjects(Integer… ids) {

//1.判定参数合法性

**if**(ids==null||ids.length==0)

**throw** **new** IllegalArgumentException("请选择一个");

//2.执行删除操作

**int** rows;

**try**{

rows=sysLogDao.deleteObjects(ids);

}**catch**(Throwable e){

e.printStackTrace();

//发出报警信息(例如给运维人员发短信)

**throw** **new** ServiceException("系统故障，正在恢复中...");

}

//4.对结果进行验证

**if**(rows==0)

**throw** **new** ServiceException("记录可能已经不存在");

//5.返回结果

**return** rows;

}

### Controller类实现

* 业务描述与设计实现

在日志控制层对象中，添加用于处理日志删除请求的方法。首先在此方法中通过形参接收客户端提交的数据，然后调用业务层对象执行删除操作，最后封装执行结果，并在运行时将响应对象转换为JSON格式的字符串，响应到客户端。

* 关键代码设计与实现

第一步：在SysLogController中添加用于执行删除业务的方法。代码如下：

@RequestMapping("doDeleteObjects")

@ResponseBody

**public** JsonResult doDeleteObjects(Integer… ids){

sysLogService.deleteObjects(ids);

**return** **new** JsonResult("delete ok");

}

第二步：启动tomcat进行访问测试，打开浏览器输入如下网址：

http://localhost/log/doDeleteObjects?ids=1,2,3

## 客户端关键业务及代码实现

### 日志列表页面事件处理

* 业务描述及设计实现

用户在页面上首先选择要删除的元素，然后点击删除按钮，将用户选择的记录id异步提交到服务端，最后在服务端执行日志的删除动作。

* 关键代码设计与实现

第一步：页面加载完成以后，在删除按钮上进行点击事件注册。关键代码如下：

...

$(".input-group-btn")

.on("click",".btn-delete",doDeleteObjects)

...

第二步：定义删除操作对应的事件处理函数。关键代码如下：

**function** doDeleteObjects(){

//1.获取选中的id值

**var** ids=doGetCheckedIds();

**if**(ids.length==0){

alert("至少选择一个");

**return**;

}

//2.发异步请求执行删除操作

**var** url="log/doDeleteObjects";

**var** params={"ids":ids.toString()};

console.log(params);

$.post(url,params,**function**(result){

**if**(result.state==1){

alert(result.message);

doGetObjects();

}**else**{

alert(result.message);

}

});

}

第三步：定义获取用户选中的记录id的函数。关键代码如下：

**function** doGetCheckedIds(){

//定义一个数组,用于存储选中的checkbox的id值

**var** array=[];//new Array();

//获取tbody中所有类型为checkbox的input元素

$("#tbodyId input[type=checkbox]").

//迭代这些元素，每发现一个元素都会执行如下回调函数

each(**function**(){

//假如此元素的checked属性的值为true

**if**($(**this**).prop("checked")){

//调用数组对象的push方法将选中对象的值存储到数组

array.push($(**this**).val());

}

});

**return** array;

}

第四步：Thead中全选元素的状态影响tbody中checkbox对象状态。代码如下：

**function** doChangeTBodyCheckBoxState(){

//1.获取当前点击对象的checked属性的值

**var** flag=$(**this**).prop("checked");//true or false

//2.将tbody中所有checkbox元素的值都修改为flag对应的值。

//第一种方案

/\* $("#tbodyId input[name='cItem']")

.each(function(){

$(this).prop("checked",flag);

}); \*/

//第二种方案

$("#tbodyId input[type='checkbox']")

.prop("checked",flag);

}

第五步：Tbody中checkbox的状态影响thead中全选元素的状态。代码如下：

**function** doChangeTHeadCheckBoxState(){

//1.设定默认状态值

**var** flag=**true**;

//2.迭代所有tbody中的checkbox值并进行与操作

$("#tbodyId input[name='cItem']")

.each(**function**(){

flag=flag&$(**this**).prop("checked")

});

//3.修改全选元素checkbox的值为flag

$("#checkAll").prop("checked",flag);

}

第六步：完善业务刷新方法，当在最后一页执行删除操作时，基于全选按钮状态及当前页码值，刷新页面。关键代码如下:

**function** doRefreshPage(){

**var** pageCount=$("#pageId").data("pageCount");

**var** pageCurrent=$("#pageId").data("pageCurrent");

**var** checked=$("#checkAll").prop("checked");

**if**(pageCurrent==pageCount&&checked&&pageCurrent>1){

pageCurrent--;

$("#pageId").data("pageCurrent",pageCurrent);

}

doGetObjects();

}

说明：最后将如上方法添加在删除操作成功以后的代码块中。

# 日志管理数据添加实现

## 服务端关键业务及代码实现

### Dao接口实现

* 业务描述与设计实现

数据层基于业务层的数据持久化请求，将业务层提交的用户行为日志信息写入到数据库。

* 关键代码设计与实现

在SysLogDao接口中添加，用于实现日志信息持久化的方法。关键代码如下：

**int** insertObject(SysLog entity);

### Mapper映射文件

* 业务描述与设计实现

基于SysLogDao中方法的定义，编写用于数据持久化的SQL元素。

* 关键代码设计与实现

在SysLogMapper.xml中添加insertObject元素，用于向日志表写入用户行为日志。关键代码如下：

<insert id="insertObject">

insert into sys\_logs

(username,operation,method,params,time,ip,createdTime)

values

(#{username},#{operation},#{method},#{params},#{time},#{ip},#{createdTime})

</insert>

### Service接口及实现类

* 业务描述与设计实现

将日志切面中抓取到的用户行为日志信息，通过业务层对象方法持久化到数据库。

* 关键代码实现

第一步：在SysLogService接口中，添加保存日志信息的方法。关键代码如下：

**void** saveObject(SysLog entity)

第二步：在SysLogServiceImpl类中添加，保存日志的方法实现。关键代码如下：

@Override

**public** **void** saveObject(SysLog entity) {

sysLogDao.insertObject(entity);

}

### 日志切面Aspect实现

* 业务描述与设计实现

在日志切面中，抓取用户行为信息，并将其封装到日志对象然后传递到业务，通过业务层对象对日志日志信息做进一步处理。此部分内容后续结合AOP进行实现（暂时先了解，不做具体实现）。

* 关键代码设计与实现

定义日志切面类对象，通过环绕通知处理日志记录操作。关键代码如下：

@Aspect

@Component

**public** **class** SysLogAspect {

**private** Logger log=Logger.*getLogger*(SysLogAspect.**class**);

@Autowired

**private** SysLogDao sysLogDao;

@Pointcut("@annotation(com.cy.pj.common.anno.RequestLog)")

**public** **void** logPointCut(){}

@Around("logPointCut()")

**public** Object around(ProceedingJoinPoint //连接点

jointPoint) **throws** Throwable{

**long** startTime=System.*currentTimeMillis*();

//执行目标方法(result为目标方法的执行结果)

Object result=jointPoint.proceed();

**long** endTime=System.*currentTimeMillis*();

**long** totalTime=endTime-startTime;

log.info("方法执行的总时长为:"+totalTime);

saveSysLog(jointPoint,totalTime);

**return** result;

}

**private** **void** saveSysLog(ProceedingJoinPoint point,

**long** totleTime) **throws** NoSuchMethodException, SecurityException, JsonProcessingException{

//1.获取日志信息

MethodSignature ms=

(MethodSignature)point.getSignature();

Class<?> targetClass=

point.getTarget().getClass();

String className=targetClass.getName();

//获取接口声明的方法

String methodName=ms.getMethod().getName();

Class<?>[] parameterTypes=ms.getMethod().getParameterTypes();

//获取目标对象方法(AOP版本不同,可能获取方法对象方式也不同)

Method targetMethod=targetClass.getDeclaredMethod(

methodName,parameterTypes);

//获取用户名,学完shiro再进行自定义实现,没有就先给固定值

String username=ShiroUtils.*getPrincipal*().getUsername();

//获取方法参数

Object[] paramsObj=point.getArgs();

System.***out***.println("paramsObj="+paramsObj);

//将参数转换为字符串

String params=**new** ObjectMapper()

.writeValueAsString(paramsObj);

//2.封装日志信息

SysLog log=**new** SysLog();

log.setUsername(username);//登陆的用户

//假如目标方法对象上有注解,我们获取注解定义的操作值

RequestLog requestLog=

targetMethod.getDeclaredAnnotation(RequestLog.**class**);

**if**(requestLog!=null){

log.setOperation(requestLog.value());

}

log.setMethod(className+"."+methodName);//className.methodName()

log.setParams(params);//method params

log.setIp(IPUtils.*getIpAddr*());//ip 地址

log.setTime(totleTime);//

log.setCreateDate(**new** Date());

//3.保存日志信息

sysLogDao.insertObject(log);

}

}

方法中用到的ip地址获取需要提供一个如下的工具类：（不用自己实现，直接用）

**public** **class** IPUtils {

**private** **static** Logger *logger* = LoggerFactory.*getLogger*(IPUtils.**class**);

**public** **static** String getIpAddr() {

HttpServletRequest request = ((ServletRequestAttributes) RequestContextHolder.*getRequestAttributes*()).getRequest();

String ip = **null**;

**try** {

ip = request.getHeader("x-forwarded-for");

**if** (StringUtils.*isEmpty*(ip) || "unknown".equalsIgnoreCase(ip)) {

ip = request.getHeader("Proxy-Client-IP");

}

**if** (StringUtils.*isEmpty*(ip) || ip.length() == 0 || "unknown".equalsIgnoreCase(ip)) {

ip = request.getHeader("WL-Proxy-Client-IP");

}

**if** (StringUtils.*isEmpty*(ip) || "unknown".equalsIgnoreCase(ip)) {

ip = request.getHeader("HTTP\_CLIENT\_IP");

}

**if** (StringUtils.*isEmpty*(ip) || "unknown".equalsIgnoreCase(ip)) {

ip = request.getHeader("HTTP\_X\_FORWARDED\_FOR");

}

**if** (StringUtils.*isEmpty*(ip) || "unknown".equalsIgnoreCase(ip)) {

ip = request.getRemoteAddr();

}

} **catch** (Exception e) {

*logger*.error("IPUtils ERROR ", e);

}

**return** ip;

}

}

原理分析，如图-11所示:

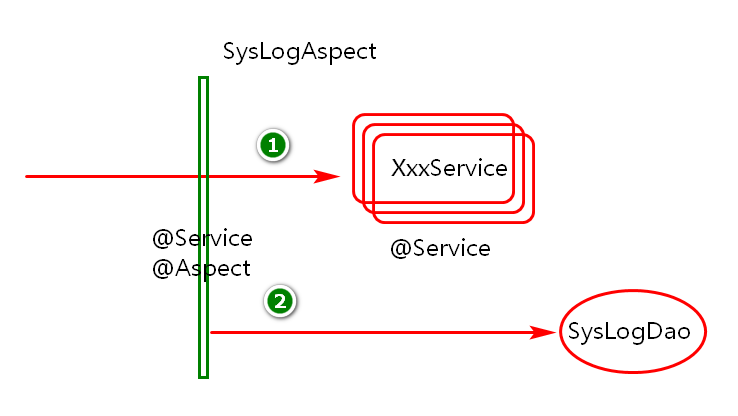


图-11

# 总结

## 重难点分析

* 日志管理整体业务分析与实现。

1. 分层架构(MVC:基于spring的mvc 模块)。
2. API架构(SysLogDao,SysLogService,SysLogController)。
3. 业务架构(查询,删除,添加用户行为日志)。
4. 数据架构(SysLog,PageObject,JsonResult,..)。

* 日志管理持久层映射文件中SQL元素的定义及编写。

1. 定义在映射文件”mapper/sys/SysLogMapper.xml”(必须在加载范围内)。
2. 每个SQL元素必须提供一个唯一ID,对于select必须指定结果映射。
3. 系统底层运行时会将每个SQL元素的对象封装一个值对象(VO)。

* 日志管理模块数据查询操作中的数据封装。

1. 数据层(数据逻辑)的SysLog对象应用(一行记录一个log对象)。
2. 业务层(业务逻辑)PageObject对象应用(封装每页记录以及对应的分页信息)。
3. 控制层(控制逻辑)的JsonResult对象应用(对业务数据添加状态信息)。

* 日志管理控制层请求数据映射，响应数据的封装及转换(转换为json 串)。

1. 请求路径映射,请求方式映射,请求参数映射。
2. 响应数据两种(页面,json串)。

* 日志管理模块异常处理如何实现的。

1. 请求处理层定义统一异常处理类。
2. 使用注解@ControllerAdvice描述类,使用@ExceptionHandler描述方法.
3. 异常处理规则:能处理则处理,不能处理则抛出。

## FAQ分析

* 用户行为日志表中都有哪些字段？
* 用户行为日志是如何实现分页查询的？
* 用户行为数据的封装过程？
* 项目中的异常是如何处理的？
* 说说的日志删除业务是如何实现？
* 如何理解 Spring MVC中的@RequestParam注解应用？
* Spring MVC 响应数据处理？
* 项目你常用的JS函数说几个？
* 日志写操作事务的传播特性如何配置?(每次开启新事务)？
* 日志写操作为什么应该是异步的?
* Spring 中的异步操作如何实现?
* Spring 中的@Async如何应用?
* 项目中的BUG分析及解决套路？

## BUG分析

* 无法找到对应的Bean对象(NoSuchBeanDefinitionException)，如图-12所示：

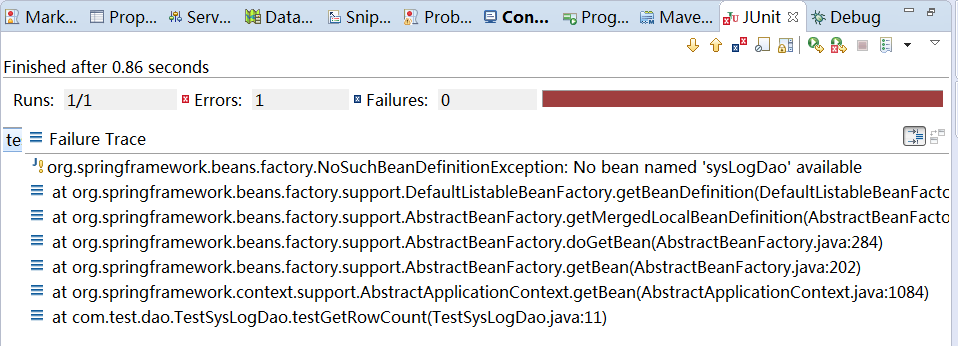


图-12

问题分析：

1. 检测key的名字写的是否正确。
2. 检测spring对此Bean对象的扫描，对于dao而言。
3. 使用有@Mapper注解描述或者在@MapperScan扫描范围之内。
4. 以上都正确，要检测是否编译了。

* 绑定异常(BindingException)，如图-13所示。

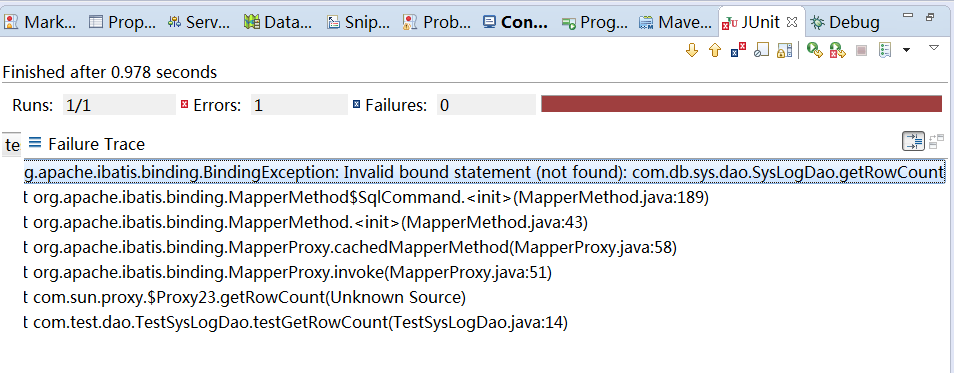


图-13

问题分析：

1. 接口的类全名与对应的映射文件命名空间不同。
2. 接口的方法名与对应的映射文件元素名不存在。
3. 检测映射文件的路径与application.properties或者application.yml中的配置是否一致。
4. 以上都没有问题时，检测你的类和映射文件是否正常编译。

* 反射异常(ReflectionException)，如图-14所示

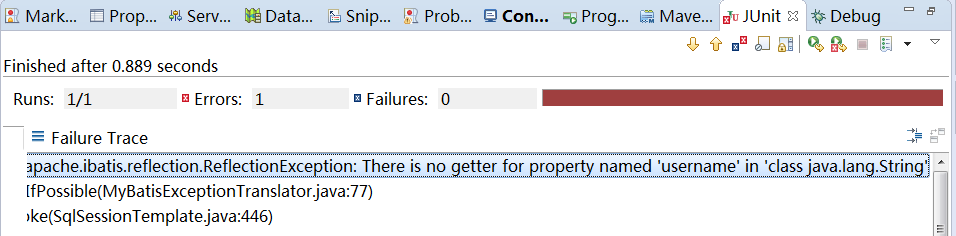


图-14

问题分析：

1. 映射文件中动态sql中使用的参数在接口方法中没有使用@Param注解修饰
2. 假如使用了注解修饰还要检测名字是否一致。

说明：当动态sql的参数在接口中没有使用@Param注解修饰，还可以借助\_parameter这个变量获取参数的值(mybatis中的一种规范)。

* Bean创建异常，如图-15所示：

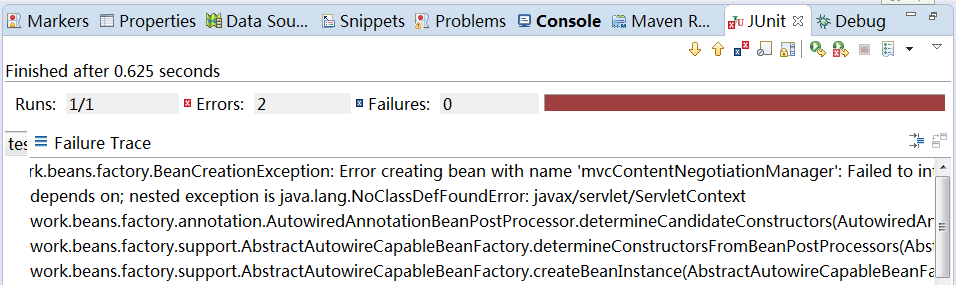


图-15

问题分析:单元测试项目缺少运行时环境(项目右键选择target runtimes)

* 结果映射异常，如图-16所示：

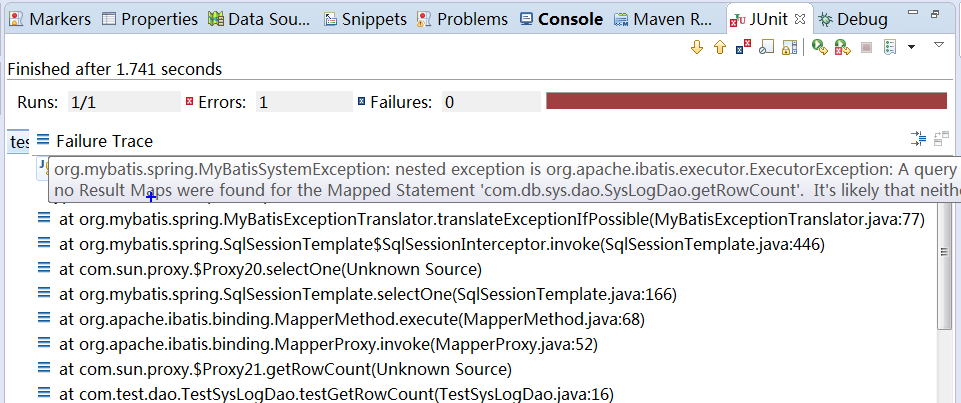


图-16

问题分析:getRowCount元素可能没有写resultType或resultMap。

* 绑定异常，如图-17所示：

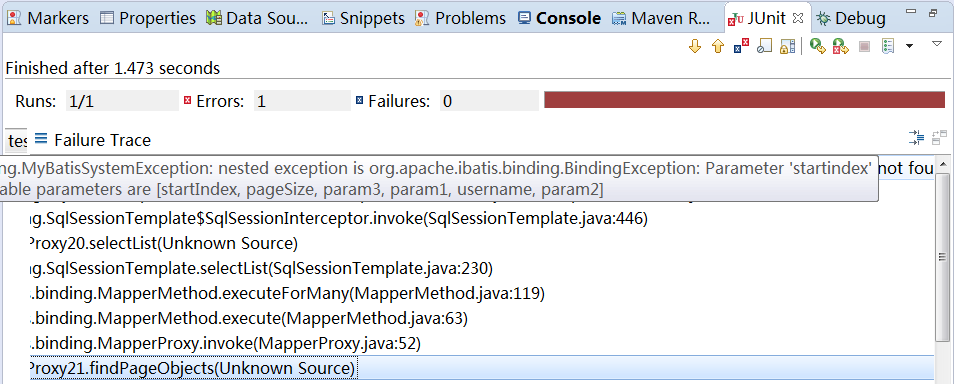


图-17

问题分析:绑定异常,检测findPageObjects方法参数与映射文件参数名字是否匹配。

* Bean创建异常，如图-18所示

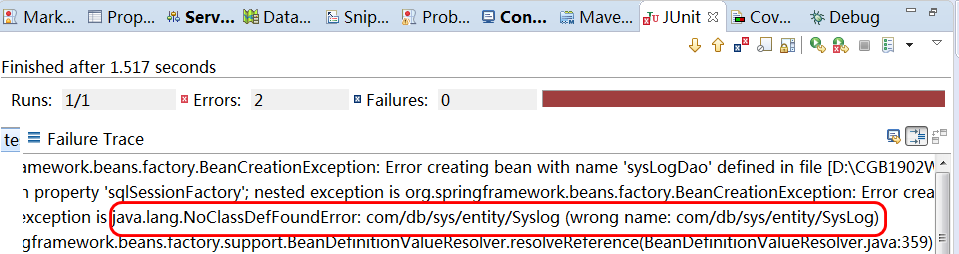


图-18

问题分析：应该是查询时的结果映射对的类全名写错了。

* 资源没找到，如图-19示

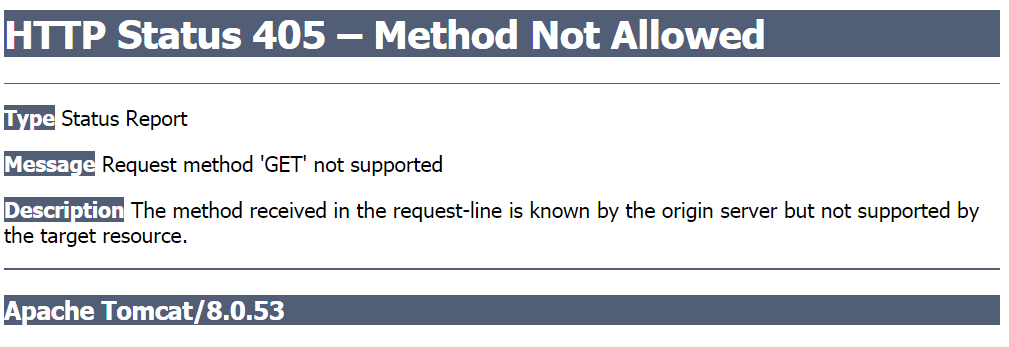


图-19

问题分析：请求方式与控制层处理方式不匹配。

* 响应结果异常，如图-20所示：

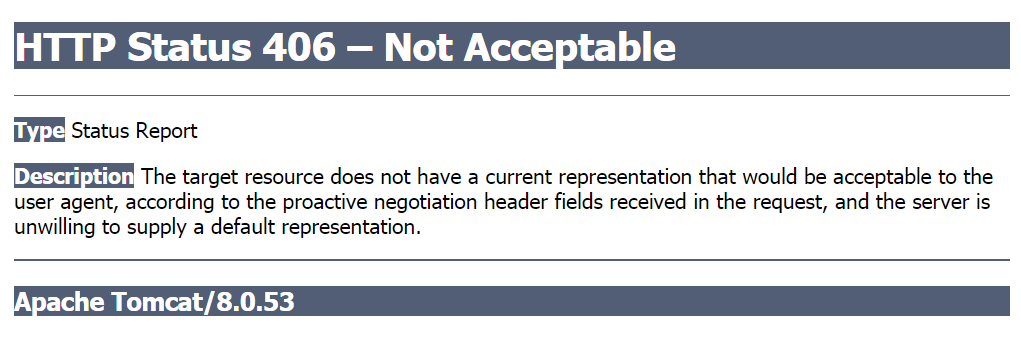


图-20

问题分析：服务端响应数据不正确，例如服务端没有注册将对象转换为JSON串的Bean

* 请求参数异常，如图-21示：

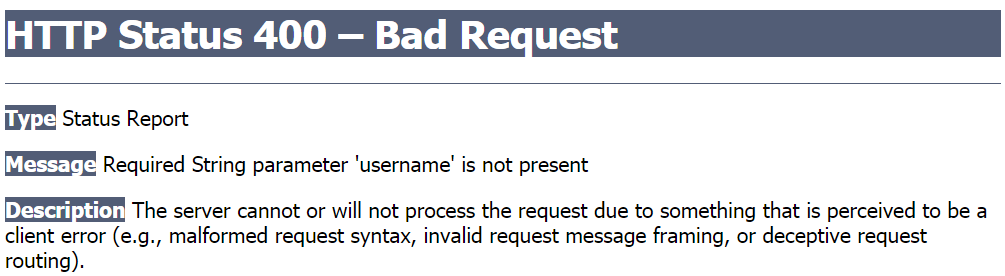


图-21

问题分析：客户端请求参数中不包含服务端控制层方法参数或格式不匹配。

* JS编写错误，如图-22所示：

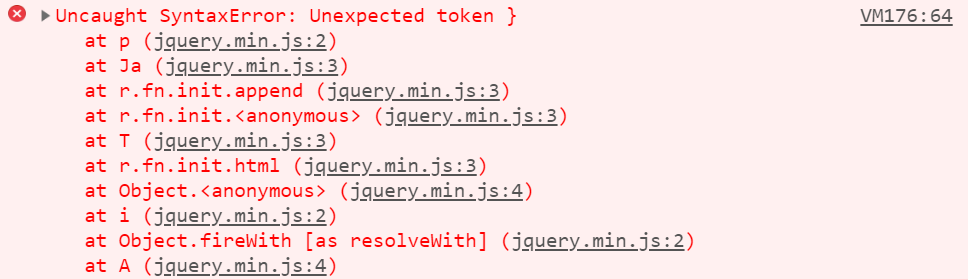


图-22

问题分析：点击右侧VM176:64位置进行查看。

* JS编写错误，如图-23所示：

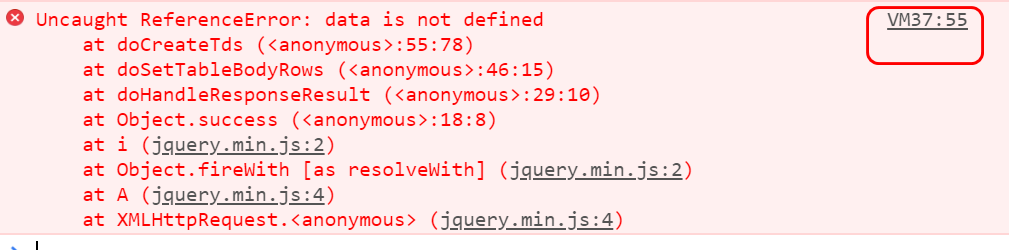


图-23

问题分析：找到对应位置，检测data的值以及数据来源。

* JS编写错误，如图-24所示：



图-24

问题分析：找到对应位置，假如无法确定位置，可排除法或打桩，debug分析。

* JS写错误，如图-25所示：

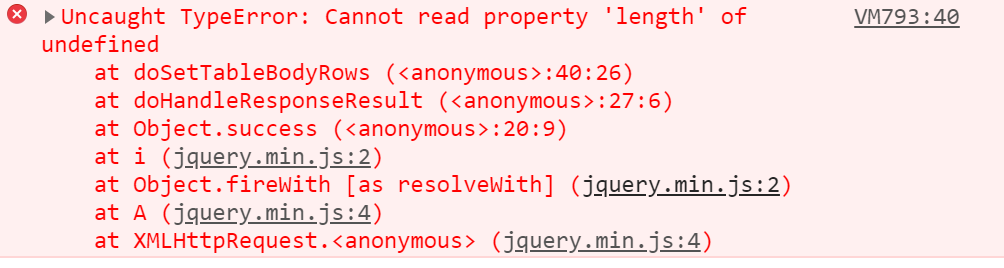


图-25

问题分析：调用length方法的对象有问题，可先检测下对象的值。

* JS编写错误，如图-26所示：



图-26

问题分析：检测record定义或赋值的地方。

* 资源没找到，如图-27所示：

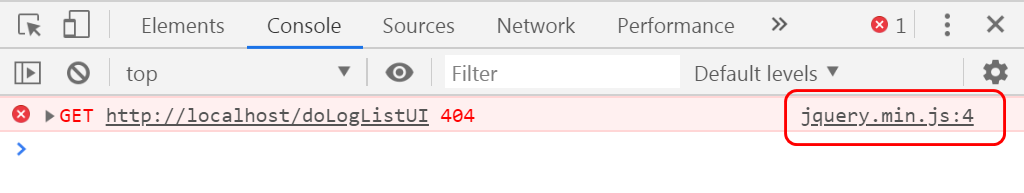


图-27

问题分析:服务端资源没找到,检查url和controller映射,不要点击图中的jquery。

* 视图解析异常，如图-28所示：

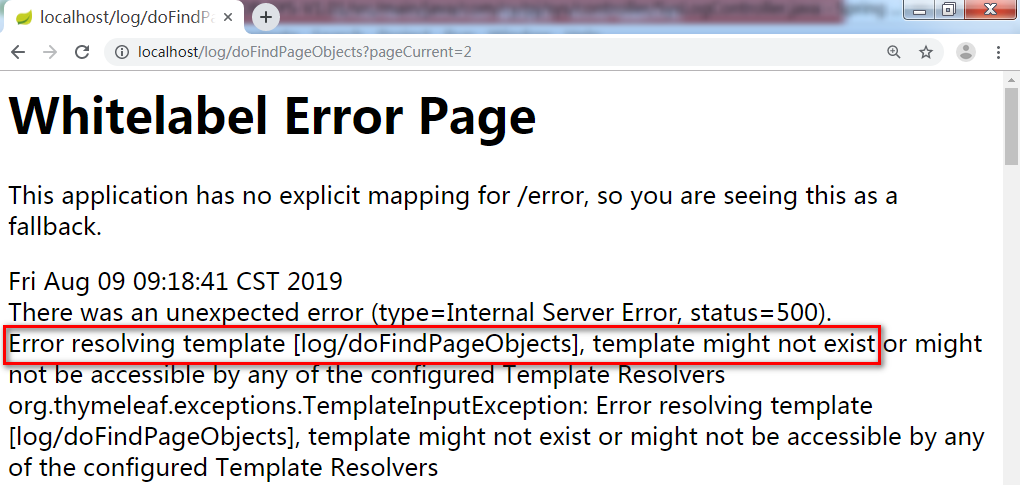


图-28

问题分析:检查服务端要访问的方法上是否有@ResponseBody注解.