后端接口

客户Controller控制器

接口一：

定义控制器接口：

```
@RestController
@RequestMapping("/customer")
```

`@RequestMapping` 注解可以应用在方法级别上，用于进一步精确匹配请求路径和请求方法。通过在方法上使用不同的 `@RequestMapping` 变体注解，可以定义不同的请求处理方法。

`@RequestMapping` 在无参数情况下，它表示将匹配所有请求路径。这意味着该方法将处理所有到达控制器的请求，而不管请求的路径是什么。

```
@Autowired
```

`@Autowired` 是一个Spring Framework提供的注解，用于自动装配（自动注入）依赖关系。当使用 `@Autowired` 注解时，Spring会自动在应用程序上下文中查找匹配类型的 bean，并将其注入到标记了 `@Autowired` 的字段、构造函数或方法参数中。

```
@RequestMapping("/getComboboxList")
@RequiresPermissions(value={"销售出库","客户退货","销售单据查询","客户退货查询","客户统计"},logical= Logical.OR)
public List<Customer> getComboboxList(String q) {
    return customerService.getComboboxList(q);
}
```

@RequiresPermissions：规定了访问权限，只有"销售出库","客户退货","销售单据查询","客户退货查询","客户统计"这些路径地方才能访问。

以下是我们后端实现接口调用的方法：

我们将customerService类的方法声明放在Service文件中：

```
public interface CustomerService {

    Map<String,Object> list(Integer page, Integer rows, String customerName);

    ServiceVO save(Customer customer);

    ServiceVO delete(String ids);

    List<Customer> getComboboxList(String q);
}
```

customerService类的方法实现放在impl文件中：

```
@Override
public List<Customer> getComboboxList(String q) {
    return customerDao.getCustomerListByNameLike(q);
}
```

我们在写程序过程中采用Dao的设计模式，用于将数据访问逻辑与业务逻辑相分离。它提供了一个抽象层，使得业务逻辑可以独立于具体的数据存储细节。我们将不同模块的细节底层逻辑封装在dao中：

```
List<Customer> getCustomerListByNameLike(@Param("q") String q);
```

相应功能的sql语句我们写在xml文件中：

```
<select id="getCustomerListByNameLike" resultMap="BaseResultMap">
    SELECT
        t_customer.customer_id,
        t_customer.customer_name,
        t_customer.contacts,
        t_customer.phone_number,
        t_customer.address,
        t_customer.remarks
    FROM
        t_customer
    <where>
        <if test="q != null and q != ''">
            t_customer.customer_name LIKE CONCAT('%', #{q} ,'%')
        </if>
    </where>
</select>
```

我们通过调用xml文件，xml文件中的sql语句获取到我们需要的数据，再将数据一层一层的返回，返回到需要用到数据的地方，这就是我们后端文件大致的结构。以下就只描述接口的功能以及返回的数据，细节就不再展开。

接口二：

```
@RequestMapping("/list")
@RequiresPermissions(value = "客户管理")// 有客户管理菜单权限的才给予调用
public Map<String,Object> list(Integer page, Integer rows, String customerName)
{
    return customerService.list(page, rows, customerName);
}
```

```
@Override
public Map<String, Object> list(Integer page, Integer rows, String customerName)
{
    Map<String,Object> map = new HashMap<>();

    page = page == 0 ? 1 : page;
    int offSet = (page - 1) * rows;

    List<Customer> customers = customerDao.getCustomerList(offSet, rows,
customerName);
```

```
    logService.save(new Log(Log.SELECT_ACTION,"分页查询客户"));

    map.put("total", customerDao.getCustomerCount(customerName));

    map.put("rows", customers);

    return map;
}
```

这是一个分页查询端口，传入参数当前页数，每页的记录数，和客户名，然后返回一个`Map<String, Object>`对象。主要目的是执行分页查询客户的操作，并将结果存储在一个`Map`对象中返回。该`Map`包含两个键值对，"total"对应客户总数，"rows"对应客户列表。如果你有任何进一步的问题，请随时提问。

接口三：

```
@RequestMapping("/save")
@RequiresPermissions(value = "客户管理")
public ServiceVO save(Customer customer) {
    return customerService.save(customer);
}
```

```
@Override
public ServiceVO save(Customer customer) {

    if(customer.getCustomerId() == null){

        customerDao.saveCustomer(customer);
        logService.save(new Log(Log.INSERT_ACTION,"添加客
户:"+customer.getCustomerName()));

    }else{

        customerDao.updateCustomer(customer);
        logService.save(new Log(Log.UPDATE_ACTION,"修改客
户:"+customer.getCustomerName()));

    }
    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

这是一个添加或者修改客户的端口，这段代码的作用是根据客户是否存在来执行相应的保存或更新操作，并将操作结果以及成功消息封装在服务值对象中返回。同时，在每个操作之后，会记录相应的日志信息以用于后续的追踪和审计。

接口四：

```
@RequestMapping("/delete")
    @RequiresPermissions(value = "客户管理")
    public ServiceVO delete(String ids) {
        return customerService.delete(ids);
    }
}
```

```
@Override
public ServiceVO delete(String ids) {

    String[] idArray = ids.split(",");

    for(String id : idArray){

        logService.save(new Log(Log.DELETE_ACTION,
                "删除客户:" +
customerDao.getCustomerById(Integer.parseInt(id)).getCustomerName()));

        customerDao.deleteCustomer(Integer.parseInt(id));

    }

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

这是一个删除客户的端口，该段代码的作用是根据输入的ID列表，逐个删除对应的客户信息。在每个删除操作之前，会记录相应的日志信息以便后续的追踪和审计。无论是否有客户被删除，方法都会返回一个成功的服务值对象。

接口五：

```
@Controller
public class DrawImageController {

    @Autowired
    private DrawImageService drawImageService;

    /**
     * 生成图片验证码
     * @param request 请求
     * @param response 响应
     */
    @GetMapping("/drawImage")
    public void drawImage(HttpServletRequest request, HttpServletResponse
response) throws Exception {
        drawImageService.drawImage(request, response);
    }

}
```

```
package com.cqu.service.impl;

import com.cqu.service.DrawImageService;
import org.springframework.stereotype.Service;

import javax.imageio.ImageIO;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.awt.*;
import java.awt.image.BufferedImage;
import java.util.Random;
```

```
@Service
public class DrawImageServiceImpl implements DrawImageService {

    private static final int WIDTH = 120;
    private static final int HEIGHT = 30;


    private static final char[] chars = {
            '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
            'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
'N',
            'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'};

    @Override
    public void drawImage(HttpServletRequest request, HttpServletResponse
response) throws Exception {
        request.setCharacterEncoding("utf-8");
        response.setContentType("text/html;charset=utf-8");
        // 创建缓存
        BufferedImage bi = new BufferedImage(WIDTH, HEIGHT,
                BufferedImage.TYPE_INT_RGB);
        // 获得画布
        Graphics g = bi.getGraphics();

        // 设置背影色
        setBackGround(g);
        // 设置边框
        setBorder(g);
        // 画干扰线
        drawRandomLine(g);
        // 写随机数
        String random = drawRandomNum((Graphics2D) g);
        // 将随机汉字存在session中
        request.getSession().setAttribute("checkcode", random);
        // 将图形写给浏览器
        response.setContentType("image/jpeg");
        // 发头控制浏览器不要缓存
        response.setDateHeader("expries", -1);
        response.setHeader("Cache-Control", "no-cache");
        response.setHeader("Pragma", "no-cache");
        // 将图片写给浏览器
        ImageIO.write(bi, "jpg", response.getOutputStream());
    }

    /**
     * 设置背景色
     *
     * @param g
     */
    private void setBackGround(Graphics g) {
        // 设置颜色
        g.setColor(new Color(22, 160, 133));
        // 填充区域
        g.fillRect(0, 0, WIDTH, HEIGHT);

    }

    /**
```

```
     * 设置边框
     *
     * @param g
     */
    private void setBorder(Graphics g) {
        // 设置边框颜色
        g.setColor(new Color(22, 160, 133));
        // 边框区域
        g.drawRect(1, 1, WIDTH - 2, HEIGHT - 2);
    }

    /**
     * 画随机线条
     *
     * @param g
     */
    private void drawRandomLine(Graphics g) {
        // 设置颜色
        g.setColor(Color.WHITE);
        // 设置线条个数并画线
        for (int i = 0; i < 5; i++) {
            int x1 = new Random().nextInt(WIDTH);
            int y1 = new Random().nextInt(HEIGHT);
            int x2 = new Random().nextInt(WIDTH);
            int y2 = new Random().nextInt(HEIGHT);
            g.drawLine(x1, y1, x2, y2);
        }

    }

    /**
     * 画随机字符
     * @param g
     * @return
     */
    private String drawRandomNum(Graphics2D g) {
        StringBuffer sb = new StringBuffer();
        // 设置颜色
        g.setColor(Color.WHITE);
        // 设置字体
        g.setFont(new Font("宋体", Font.BOLD, 20));

        int x = 5;
        // 控制字数
        for (int i = 0; i < 4; i++) {
            // 设置字体旋转角度
            int degree = new Random().nextInt() % 30;
            // 截取随机字符索引
            int n = new Random().nextInt(chars.length);
            sb.append(chars[n]);
            // 正向角度
            g.rotate(degree * Math.PI / 180, x, 20);
            g.drawString(chars[n] + "", x, 20);
            // 反向角度
            g.rotate(-degree * Math.PI / 180, x, 20);
            x += 30;
        }
        return sb.toString();
```

```
    }
}
```

这个是验证码的接口，定义验证码框的大小和颜色，字体的样式和颜色，以及背景的颜色。并在验证框画干扰线。

接口六:

```
@RequestMapping("/listInventory")
@RequiresPermissions(value="当前库存查询")
public Map<String,Object> listInventory(Integer page, Integer rows, String
codeOrName, Integer goodsTypeId) {
    return goodsService.listInventory(page, rows, codeOrName, goodsTypeId);
}
```

```
@Override
public Map<String, Object> listInventory(Integer page, Integer rows, String
codeOrName, Integer goodsTypeId) {
    Map<String,Object> map = new HashMap<>();

    page = page == 0 ? 1 : page;
    int offSet = (page - 1) * rows;

    List<Goods> goodsList = goodsDao.getGoodsInventoryList(offSet, rows,
codeOrName, goodsTypeId);

    for (Goods goods : goodsList) {
        // 销售总量等于销售单据的销售数据减去退货单据的退货数据

goods.setSaleTotal(saleListGoodsService.getSaleTotalByGoodsId(goods.getGoodsId()
)
                );

    }

    map.put("rows", goodsList);

    map.put("total", goodsDao.getGoodsInventoryCount(codeOrName, goodsTypeId));

    logService.save(new Log(Log.SELECT_ACTION, "分页查询商品库存信息"));

    return map;
}
```

这是一个查询当前库存的端口，它接受页码（page）、每页行数（rows）、商品编码或名称（codeOrName）和商品类型ID（goodsTypeId）作为参数，并返回一个包含商品库存信息的Map对象。

接口七:

```
@RequestMapping("/list")
@RequiresPermissions(value={"商品管理","进货入库","退货出库","销售出库","客户退货","商品报损","商品报溢"},logical= Logical.OR)
public Map<String,Object> list(Integer page, Integer rows, String goodsName, Integer goodsTypeId) {
    return goodsService.list(page, rows, goodsName, goodsTypeId);
}
```

```
@Override
public Map<String, Object> list(Integer page, Integer rows, String goodsName, Integer goodsTypeId) {
    Map<String,Object> map = new HashMap<>();

    page = page == 0 ? 1 : page;
    int offSet = (page - 1) * rows;

    // 查询类别ID为当前ID或父ID为当前类别ID的商品
    List<Goods> goodsList = goodsDao.getGoodsList(offSet, rows, goodsName, goodsTypeId);

    map.put("rows", goodsList);

    map.put("total", goodsDao.getGoodsCount(goodsName, goodsTypeId));

    logService.save(new Log(Log.SELECT_ACTION, "分页查询商品信息"));

    return map;
}
```

这是一个分页查询的端口，它接收一些参数（`page`、`rows`、`goodsName`、`goodsTypeId`）并返回一个包含商品信息的`Map`对象。通过`page`、`rows`、`goodsName`、`goodsTypeId`来查询到对应的商品，并将商品添加到map中，最后把map放回。

接口八：

```
@RequestMapping("/getCode")
@RequiresPermissions(value = "商品管理")
public ServiceVO getCode() {
    return goodsService.getCode();
}
```

```
@Override
public ServiceVO getCode() {

    // 获取当前商品最大编码
    String code = goodsDao.getMaxCode();

    // 在现有编码上加1
    Integer intCode = Integer.parseInt(code) + 1;

    // 将编码重新格式化为4位数字符串形式
    String unitCode = intCode.toString();

    for(int i = 4;i > intCode.toString().length();i--){
```

```
        unitCode = "0"+unitCode;

    }
    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS,
unitCode);
}
```

这是一个商品编码端口，给每个商品生成他对应的编码，在原有的编码上面实现自增。

接口九：

```
@RequestMapping("/save")
@ResponseBody
@RequiresPermissions(value = "商品管理")
public ServiceVO save(Goods goods) {
    return goodsService.save(goods);
}
```

```
@Override
public ServiceVO save(Goods goods) {

    if(goods.getGoodsId() == null){

        logService.save(new Log(Log.INSERT_ACTION,"添加商
品:"+goods.getGoodsName()));
        // 设置上一次采购价为当前采购价
        goods.setLastPurchasingPrice(goods.getPurchasingPrice());
        goods.setInventoryQuantity(0);
        goods.setState(0);
        goodsDao.saveGoods(goods);

    }else{

        goodsDao.updateGoods(goods);
        logService.save(new Log(Log.UPDATE_ACTION,"修改商
品:"+goods.getGoodsName()));

    }

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);

}
```

这是一个添加或修改商品信息的端口，并将修改或添加操作存入日志中，最后返回状态码。

接口十：

```
@RequestMapping("/getNoInventoryQuantity")
@RequiresPermissions(value = "期初库存")
public Map<String,Object> getNoInventoryQuantity(Integer page,Integer rows,String
nameOrCode) {
    return goodsService.getNoInventoryQuantity(page, rows, nameOrCode);
}
```

```
@Override
public Map<String, Object> getNoInventoryQuantity(Integer page, Integer rows,
String nameOrCode) {
    Map<String,Object> map = new HashMap<>();

    page = page == 0 ? 1 : page;
    int offSet = (page - 1) * rows;

    List<Goods> goodsList = goodsDao.getNoInventoryQuantity(offSet, rows,
nameOrCode);

    map.put("rows", goodsList);

    map.put("total", goodsDao.getNoInventoryQuantityCount(nameOrCode));

    logService.save(new Log(Log.SELECT_ACTION, "分页查询商品信息（无库存）"));

    return map;
}
```

这是一个分页查询已有库存商品信息的端口，使用`goodsDao`对象调用`getNoInventoryQuantity`方法查询没有库存数量的商品列表，该方法接收偏移量、行数和商品名称/编码作为参数，并返回商品列表。将商品列表添加到`map`中，使用键名"rows"。继续添加键值对到`map`中，调用`goodsDao`的`getNoInventoryQuantityCount`方法获取符合条件的没有库存数量的商品总数，并使用键名"total"将其添加到`map`中。之后，调用`logService`对象的`save`方法，将一条日志保存到数据库中，记录执行了"分页查询商品信息（无库存）"操作。最后，返回填充完数据的`map`对象作为方法的结果。

接口十一：

```
@RequestMapping("/saveStock")
    @ResponseBody
    @RequiresPermissions(value = "期初库存")
    public ServiceVO saveStock(Integer goodsId,Integer inventoryQuantity,double
purchasingPrice) {
        return goodsService.saveStock(goodsId, inventoryQuantity,
purchasingPrice);
    }
```

```
@Override
    public ServiceVO saveStock(Integer goodsId, Integer inventoryQuantity, double
purchasingPrice) {

        Goods goods = goodsDao.findByGoodsId(goodsId);

        goods.setInventoryQuantity(inventoryQuantity);

        goods.setPurchasingPrice(purchasingPrice);

        goods.setLastPurchasingPrice(purchasingPrice);

        goodsDao.updateGoods(goods);

        logService.save(new Log(Log.UPDATE_ACTION,goods.getGoodsName()+"商品期初入
库"));
```

```
        return new ServiceVO<>(SuccessCode.SUCCESS_CODE,
SuccessCode.SUCCESS_MESS);
    }
```

这是一个添加商品的端口。首先，通过调用`goodsDao`对象的`findByGoodsId`方法，根据商品ID获取对应的商品对象。然后，将商品对象的库存数量（`inventoryQuantity`）设置为传入的库存数量参数。接着，将商品对象的采购价格（`purchasingPrice`）设置为传入的采购价格参数。同时，将商品对象的上一次采购价（`lastPurchasingPrice`）也设置为传入的采购价格参数。接下来，通过调用`goodsDao`对象的`updateGoods`方法，将更新后的商品对象保存到数据库中。然后，调用`logService`对象的`save`方法，将一条日志保存到数据库中，记录执行了"商品名称商品期初入库"的操作，其中商品名称从商品对象的`getGoodsName()`方法获取。最后，创建一个`ServiceVO`对象，并将成功的状态码（`SuccessCode.SUCCESS_CODE`）和成功的消息（`SuccessCode.SUCCESS_MESS`）设置进去。最终，返回填充好的`ServiceVO`对象作为方法的结果。

接口十二：

```
@RequestMapping("/deleteStock")
@ResponseBody
@RequiresPermissions(value = "期初库存")
public ServiceVO deleteStock(Integer goodsId) {
    return goodsService.deleteStock(goodsId);
}
```

```
@Override
public ServiceVO deleteStock(Integer goodsId) {

    Goods goods = goodsDao.findByGoodsId(goodsId);

    if (goods.getState() == 2) {

        return new ServiceVO<>(ErrorCode.HAS_FORM_ERROR_CODE,
ErrorCode.HAS_FORM_ERROR_MESS);
    }

    goods.setInventoryQuantity(0);
    goodsDao.updateGoods(goods);

    logService.save(new Log(Log.UPDATE_ACTION,goods.getGoodsName()+"商品清除库
存"));

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

接口十三：

```
@RequestMapping("/listAlarm")
@RequiresPermissions(value = "库存报警")
public Map<String,Object> listAlarm() {
    return goodsService.listAlarm();
}
```

```
@Override
public Map<String, Object> listAlarm() {
    Map<String,Object> map = new HashMap<>();

    List<Goods> goodsList = goodsDao.getGoodsAlarm();

    map.put("rows", goodsList);

    logService.save(new Log(Log.SELECT_ACTION, "查询库存报警商品信息"));

    return map;
}
```

这是一个查询库存的端口，通过调用`goodsDao.getGoodsAlarm()`方法从数据库中获取库存报警商品列表。将获取到的商品列表存储在`map`对象中，使用键名"rows"来关联。这样做可以方便通过键名来取得商品列表。最后，返回存储有商品列表的`map`对象作为方法的结果。

接口十四：

```
@RequestMapping("/delete")
@RequiresPermissions(value = {"商品管理","进货入库","退货出库","销售出库","客户退货","商品报损","商品报溢"},logical=Logical.OR)
public ServiceVO delete(Integer goodsTypeId) {
    return goodsTypeService.delete(goodsTypeId);
}
```

```
@Override
public ServiceVO delete(Integer goodsTypeId) {

    // 根据商品类别ID来查询商品信息，如果该类别下有商品信息，则不给予删除
    List<Goods> goodsList = goodsTypeDao.getGoodsByTypeId(goodsTypeId);

    if (goodsList.size() != 0) {

        return new ServiceVO<>(ErrorCode.GOODS_TYPE_ERROR_CODE,
ErrorCode.GOODS_TYPE_ERROR_MESS);
    }

    // 这里的逻辑是先根据商品类别ID查询出商品类别的信息，找到商品类别的父级商品类别
    // 如果父商品类别的子商品类别信息等于1，那么再删除商品信息的时候，父级商品类别的状态也应该从根节点改为叶子节点
    GoodsType goodsType = goodsTypeDao.getGoodsTypeById(goodsTypeId);

    List<GoodsType> goodsTypeList =
goodsTypeDao.getAllGoodsTypeByParentId(goodsType.getPId());

    if(goodsTypeList.size() == 1){

        GoodsType parentGoodsType =
goodsTypeDao.getGoodsTypeById(goodsType.getPId());

        parentGoodsType.setGoodsTypeState(0);

        goodsTypeDao.updateGoodsTypeState(parentGoodsType);
```

```
    }

    goodsTypeDao.delete(goodsTypeId);

    logService.save(new Log(Log.DELETE_ACTION,"删除商品类
别："+goodsType.getGoodsTypeName()));

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);

}
```

这是一个删除商品类别的端口，如果商品信息列表的大小为0，即该商品类别下没有任何商品信息，继续执行下面的逻辑。通过调用`getGoodsTypeById`方法，根据商品类别ID获取商品类别实例（`goodsType`）。再通过调用`getAllGoodsTypeByParentId`方法，获取与该商品类别具有相同父级的所有商品类别（`goodsTypeList`）。检查商品类别列表的大小是否为1，如果为1，表示当前商品类别是其父商品类别的唯一子类别。在这种情况下，通过调用`getGoodsTypeById`方法，获取父商品类别的实例（`parentGoodsType`）。将父商品类别的状态设置为叶子节点（将状态设置为0）。通过调用`updateGoodsTypeState`方法，更新父商品类别的状态。调用`delete`方法，根据商品类别ID删除商品类别信息。通过调用`save`方法，将一条删除商品类别的操作记录保存到日志中。1最后，返回一个包含成功代码和成功消息的`ServiceVO`对象，表示操作成功完成。

接口十五：

```
@RequestMapping("/save")
@RequiresPermissions(value={"商品管理","进货入库","退货出库","销售出库","客户退货","商品
报损","商品报溢"},logical = Logical.OR)
public ServiceVO save(String goodsTypeName,Integer pId) {
    return goodsTypeService.save(goodsTypeName, pId);
}
```

```
@Override
public ServiceVO save(String goodsTypeName, Integer pId) {

    GoodsType goodsType = new GoodsType(goodsTypeName,0,pId);

    goodsTypeDao.saveGoodsType(goodsType);

    // 根据父类ID来查询父类实体
    GoodsType parentGoodsType = goodsTypeDao.getGoodsTypeById(pId);

    // 如果当前父商品类别是叶子节点，则需要修改为状态为根节点
    if(parentGoodsType.getGoodsTypeState() == 0) {

        parentGoodsType.setGoodsTypeState(1);

        goodsTypeDao.updateGoodsTypeState(parentGoodsType);

    }

    logService.save(new Log(Log.INSERT_ACTION, "新增商品类别:"+goodsTypeName));

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

这是一个添加商品类别的端口，首先，根据传入的商品类别名称（goodsTypeName）和父类别ID（pId），创建一个新的商品类别实例（`goodsType`）。该实例的状态设置为0，表示默认为叶子节点。通过调用`saveGoodsType`方法，将商品类别实例保存到数据库中。再通过调用`getGoodsTypeById`方法，根据父类别ID获取父类别的实例（`parentGoodsType`）。检查父商品类别的状态是否为0，如果为0，表示当前父商品类别是叶子节点。在这种情况下，将父商品类别的状态修改为根节点（将状态设置为1）。通过调用`updateGoodsTypeState`方法，更新父商品类别的状态。调用`save`方法，将一条新增商品类别的操作记录保存到日志中。最后，返回一个包含成功代码和成功消息的`ServiceVO`对象，表示操作成功完成。

接口十六:

```
@RequestMapping("/loadGoodsType")
@RequiresPermissions(value={"商品管理","进货入库","退货出库","销售出库","客户退货","当前库存查询","商品报损","商品报溢","商品采购统计"},logical = Logical.OR)
public String loadGoodsType() {
    return goodsTypeService.loadGoodsType();
}
```

```
@Override
public String loadGoodsType() {
    logService.save(new Log(Log.SELECT_ACTION, "查询商品类别信息"));

    return this.getAllGoodsType(-1).toString(); // 根节点默认从-1开始
}
```

这是一个查询商品所有类别的端口，首先，通过调用`save`方法，将一条查询商品类别信息的操作记录保存到日志中。调用`getAllGoodsType`方法，并传入参数-1，获取所有商品类别的信息。这里的-1表示从根节点开始获取。将获取到的商品类别信息转换为字符串表示（使用`.toString()`方法）。最后，方法返回表示商品类别信息的字符串。

接口十七:

```
@GetMapping("/")
public String toIndex(){
    return "redirect:login.html";
}
```

这个接口的作用是将用户请求重定向到"login.html"页面。当用户访问根路径"/"时，服务器会发送一个重定向响应，告诉浏览器跳转到"login.html"页面。这样可以实现用户在访问网站时自动跳转到登录页面。

接口十八:

```
@RequestMapping("/list")
public String list(String logType,String trueName,String sTime,String eTime,Integer page,Integer rows) {
    return logService.list(logType, trueName, sTime, eTime, page, rows);
}
```

```
@Override
public String list(String logType, String trueName, String sTime, String eTime, Integer page, Integer rows) {
```

```
    JsonObject result = new JsonObject();

    JsonArray array = new JsonArray();

    try {
        // 测试
        System.err.println(eTime);
        System.err.println(DateUtil.StringToDate(eTime, "yyyy-MM-dd HH:mm:ss"));

        page = page == 0 ? 1 : page;
        int offSet = (page - 1) * rows;

        List<Log> logList = logDao.getLogList(logType, trueName,
DateUtil.StringToDate(sTime, "yyyy-MM-dd HH:mm:ss"),
                DateUtil.StringToDate(eTime, "yyyy-MM-dd HH:mm:ss"), offSet,
rows);

        for(Log log : logList) {

            JsonObject obj = new JsonObject();

            obj.addProperty("logId", log.getLogId());

            obj.addProperty("logType", log.getLogType());

            obj.addProperty("trueName",
logDao.getTrueNameByLogId(log.getLogId()));

            obj.addProperty("logDate", DateUtil.DateToString(log.getLogDate(),
"yyyy-MM-dd HH:mm:ss"));

            obj.addProperty("content", log.getContent());

            array.add(obj);

        }

        Long total = logDao.getLogCount(logType, trueName,
DateUtil.StringToDate(sTime, "yyyy-MM-dd HH:mm:ss"),
                DateUtil.StringToDate(eTime, "yyyy-MM-dd HH:mm:ss"));

        result.add("rows", array);

        result.addProperty("total", total);


    } catch (ParseException e) {

        e.printStackTrace();
    }

    return result.toString();
}
```

这是一个日志接口，它接受一些参数，并返回一个包含日志信息的JSON字符串。该方法的签名表明它是一个重写（`@Override`）的方法，所以它应该是属于某个接口或父类中的一个方法。在方法的实现中，首先创建了一个`JsonObject`对象`result`，用于存储最终的结果。接下来创建了一个空的`JsonArray`对象`array`，用于存储日志信息的数组。在try-catch块内部，对输入的时间参数进行处理和转换，将其解析成`Date`对象，这里使用了一个自定义的`DateUtil`工具类。然后根据传入的参数调用`logDao`对象的方法`getLogList`，获取满足条件的日志列表。这个方法可能会从数据库中查询相应的数据。对于每个日志对象，在循环中创建一个`JsonObject`对象`obj`，并将日志的各个属性添加到该对象中。将这个`obj`对象添加到之前创建的`array`数组中。通过调用`logDao`对象的方法`getLogCount`获取总日志数，用于分页显示。将最终的`rows`数组和`total`值添加到`result`对象中。最后，在catch块中处理异常情况。

接口十九：

```
@RequestMapping("/loadMenu")
public String loadMenu(HttpSession session) {
    return menuService.loadMenu(session);
}
```

```
@Override
public String loadMenu(HttpSession session) {
    // 获取登录中的角色
    Role currentRole = (Role) session.getAttribute("currentRole");

    return this.getAllMenu(-1,currentRole.getRoleId()).toString();// 根节点默认从-1
开始
}
```

这是一个查询当前角色的导航菜单端口，首先通过调用`session.getAttribute("currentRole")`从会话中获取当前登录用户的角色对象，并将其存储在`currentRole`变量中。接下来，通过调用`this.getAllMenu(-1, currentRole.getRoleId())`来获取与当前角色ID相关联的所有菜单信息。`getAllMenu(-1, currentRole.getRoleId())`是类中的另一个方法，它接受两个参数：根节点的ID（此处为-1）和当前角色的ID。返回的菜单信息可能是一个对象或者集合。最后，通过调用`.toString()`方法将菜单信息转换为字符串形式，并将其作为结果返回给调用方。

接口二十：

```
@RequestMapping("/loadCheckMenu")
@RequiresPermissions(value = "角色管理")
public String loadCheckMenu(Integer roleId) {
    return menuService.loadCheckMenu(roleId);
}
```

```
@Override
public String loadCheckMenu(Integer roleId) {
    List<Menu> menuList = menuDao.getMenuByRoleId(roleId);

    List<Integer> menuIds = new ArrayList<>();

    for(Menu menu : menuList) {

        menuIds.add(menu.getMenuId());
```

```
    }

    logService.save(new Log(Log.SELECT_ACTION, "查询菜单信息"));

    return this.getCheckAllMenu(-1,menuIds).toString(); //根节点默认从-1开始
}
```

这是一个当前角色菜单接口，首先通过调用`menuDao.getMenuByRoleId(roleId)`获取与给定角色ID相关联的菜单列表。然后，创建了一个空的`ArrayList`对象`menuIds`，用于存储菜单的ID。接下来，使用`for`循环遍历`menuList`中的每个菜单对象。在循环中，通过调用`menu.getMenuId()`获取当前菜单的ID，并将其添加到`menuIds`列表中。随后，调用`logService.save(new Log(Log.SELECT_ACTION, "查询菜单信息"))`以保存一个日志记录，表示执行了查询菜单的操作。最后，通过调用`this.getCheckAllMenu(-1, menuIds).toString()`来获取所有菜单的信息，并将其转换为字符串形式进行返回。`getCheckAllMenu(-1, menuIds)`是一个类中的另一个方法，它接受两个参数：根节点的ID（此处为-1）和菜单ID列表。返回的菜单信息可能是一个对象或者集合，通过调用`.toString()`方法将其转换为字符串形式返回给调用方。

接口二十一：

```
@RequestMapping("/save")
@RequiresPermissions(value = "进货入库")
public ServiceVO save(PurchaseList purchaseList, String purchaseListGoodsStr) {
    return purchaseListGoodsService.save(purchaseList, purchaseListGoodsStr);
}
```

```
@Override
public ServiceVO save(PurchaseList purchaseList, String purchaseListGoodsStr) {

    // 使用谷歌Gson将JSON字符串数组转换成具体的集合
    Gson gson = new Gson();

    List<PurchaseListGoods> purchaseListGoodsList =
gson.fromJson(purchaseListGoodsStr,new TypeToken<List<PurchaseListGoods>>()
{}.getType());

    // 设置当前操作用户
    User currentUser = userDao.findUserByName((String)
SecurityUtils.getSubject().getPrincipal());

    purchaseList.setUserId(currentUser.getUserId());

    // 保存进货清单
    purchaseListGoodsDao.savePurchaseList(purchaseList);

    // 保存进货商品列表
    for (PurchaseListGoods p : purchaseListGoodsList) {
        p.setPurchaseListId(purchaseList.getPurchaseListId());

p.setGoodsTypeId(goodsDao.findByGoodsId(p.getGoodsId()).getGoodsTypeId());
        purchaseListGoodsDao.savePurchaseListGoods(p);

        // 修改商品上一次进货价，进货均价，库存，状态
        Goods goods = goodsDao.findByGoodsId(p.getGoodsId());

        goods.setLastPurchasingPrice(p.getPrice());
```

```
        goods.setInventoryQuantity(goods.getInventoryQuantity() +
p.getGoodsNum());


goods.setPurchasingPrice(BigDecimalUtil.keepTwoDecimalPlaces((goods.getPurchasin
gPrice()+p.getPrice())/2));

        goods.setState(2);

        goodsDao.updateGoods(goods);
    }

    // 保存日志
    logService.save(new Log(Log.INSERT_ACTION, "新增进货
单："+purchaseList.getPurchaseNumber()));

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

这是一个保存进货信息单的接口，实现了保存进货清单和商品列表的功能，并对相应的商品进行更新和记录日志。

接口二十二：

```
@RequestMapping("/list")
@RequiresPermissions(value={"进货单据查询","供应商统计"},logical = Logical.OR)
public Map<String,Object> list(String purchaseNumber, Integer supplierId, Integer
state, String sTime,
                               String eTime) {
    return  purchaseListGoodsService.list(purchaseNumber, supplierId, state,
sTime, eTime);
}
```

```
@Override
public Map<String, Object> list(String purchaseNumber, Integer supplierId,
Integer state, String sTime, String eTime) {
    Map<String,Object> result = new HashMap<>();


    List<PurchaseList> purchaseListList =
purchaseListGoodsDao.getPurchaselist(purchaseNumber, supplierId, state, sTime,
eTime);

    logService.save(new Log(Log.SELECT_ACTION, "进货单据查询"));

    result.put("rows", purchaseListList);

    return result;
}
```

这是一个查询进货单的接口，实现了根据给定的条件查询进货单据的功能，并将查询结果以Map形式返回。查询结果可以通过"rows"键获取。

接口二十三：

```
@RequestMapping("/goodsList")
@RequiresPermissions(value={"进货单据查询","供应商统计"},logical = Logical.OR)
public Map<String, Object> goodsList(Integer purchaseListId) {
    return purchaseListGoodsService.goodsList(purchaseListId);
}
```

```
@Override
public Map<String, Object> goodsList(Integer purchaseListId) {
    Map<String, Object> map = new HashMap<>();

    List<PurchaseListGoods> purchaseListGoodsList =
purchaseListGoodsDao.getPurchaseListGoodsByPurchaseListId(purchaseListId);

    logService.save(new Log(Log.SELECT_ACTION, "进货单商品信息查询"));

    map.put("rows", purchaseListGoodsList);

    return map;
}
```

这是一个查询进货单商品信息接口，实现了根据给定的进货单据ID查询其商品信息的功能，并将查询结果以Map形式返回。查询结果可以通过"rows"键获取。

接口二十四：

```
@RequestMapping("/delete")
@RequiresPermissions(value = "进货单据查询")
public ServiceVO delete(Integer purchaseListId) {
    return purchaseListGoodsService.delete(purchaseListId);
}
```

```
@Override
public ServiceVO delete(Integer purchaseListId) {

    logService.save(new Log(Log.DELETE_ACTION, "删除进货
单："+purchaseListGoodsDao.getPurchaseListById(purchaseListId).getPurchaseNumber(
)));


purchaseListGoodsDao.deletePurchaseListGoodsByPurchaseListId(purchaseListId);

    purchaseListGoodsDao.deletePurchaseListById(purchaseListId);

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

这是一个删除进货单及商品信息接口，实现了删除特定进货单据及其相关商品信息的功能，并在数据库中执行相应的删除操作。最后，返回一个表示成功的ServiceVO对象。

接口二十五：

```
@RequestMapping("/updateState")
@RequiresPermissions(value = "供应商统计")
public ServiceVO updateState(Integer purchaseListId) {
    return purchaseListGoodsService.updateState(purchaseListId);
}
```

```
@Override
public ServiceVO updateState(Integer purchaseListId) {
    purchaseListGoodsDao.updateState( purchaseListId);

    logService.save(new Log(Log.DELETE_ACTION, "支付结算进货
单："+purchaseListGoodsDao.getPurchaseListById(purchaseListId).getPurchaseNumber(
)));

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

这是一个修改进货单付款状态接口，实现了将特定进货单据的状态更新为支付结算的功能，并在数据库中执行相应的更新操作。最后，返回一个表示成功的ServiceVO对象。

接口二十六：

```
@RequestMapping("/count")
@RequiresPermissions(value = "商品采购统计")
public String count(String sTime, String eTime ,Integer goodsTypeId, String
codeOrName) {
    return purchaseListGoodsService.count(sTime, eTime, goodsTypeId,
codeOrName);
}
```

```
@Override
public String count(String sTime, String eTime, Integer goodsTypeId, String
codeOrName) {

    JsonArray result = new JsonArray();

    try {

        List<PurchaseList> purchaseListList =
purchaseListGoodsDao.getPurchaselist(null, null, null, sTime, eTime);

        for(PurchaseList pl : purchaseListList){

            List<PurchaseListGoods> purchaseListGoodsList = purchaseListGoodsDao
                    .getPurchaseListGoods(pl.getPurchaseListId(), goodsTypeId,
codeOrName);

            for(PurchaseListGoods pg : purchaseListGoodsList){

                JsonObject obj = new JsonObject();

                obj.addProperty("number", pl.getPurchaseNumber());
```

```
                obj.addProperty("date", pl.getPurchaseDate());

                obj.addProperty("supplierName", pl.getSupplierName());

                obj.addProperty("code", pg.getGoodsCode());

                obj.addProperty("name", pg.getGoodsName());

                obj.addProperty("model", pg.getGoodsModel());

                obj.addProperty("goodsType",
goodsTypeDao.getGoodsTypeById(pg.getGoodsTypeId()).getGoodsTypeName());

                obj.addProperty("unit", pg.getGoodsUnit());

                obj.addProperty("price", pg.getPrice());

                obj.addProperty("num", pg.getGoodsNum());

                obj.addProperty("total", pg.getTotal());

                result.add(obj);

            }
        }

        logService.save(new Log(Log.SELECT_ACTION, "进货商品统计查询"));

    } catch (Exception e) {

        e.printStackTrace();

    }

    return result.toString();
}
```

这是一个进货商品统计接口，实现了根据给定的时间范围、商品类型和编码/名称查询进货商品统计信息的功能，并将查询结果以JSON格式返回。每个进货商品的相关信息被封装为一个JsonObject对象，并添加到JsonArray对象中。最后，返回JsonArray对象的字符串表示。

接口二十七：

```
@PostMapping("/saveRole")
public ServiceVO saveRole(@RequestBody Role role, HttpSession session) {
    return roleService.saveRole(role, session);
}
```

```
@Override
public ServiceVO saveRole(Role role, HttpSession session) {
    User user = userDao.findUserByName((String)
SecurityUtils.getSubject().getPrincipal());
    Role roleDB = roleDao.getRoleByRoleIdUserId(role.getRoleId(),
user.getUserId());
    // 将用户选择的角色信息放入缓存，以便以后加载权限的时候使用
    session.setAttribute("currentRole", roleDB);
    System.out.println("db the role --->" + roleDB);
    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

这是一个保存用户登录时所选择的角色接口，这段代码的功能是根据用户选择的角色信息，从数据库中获取对应的角色对象，并将其存储在会话对象中。同时，它也输出了获取到的角色对象到控制台。最后，它返回一个包含成功代码和成功消息的`ServiceVO`对象作为方法的结果。

接口二十八：

```
@RequestMapping("/listAll")
@RequiresPermissions(value = {"用户管理","角色管理"}, logical = Logical.OR)
public Map<String,Object> listAll() {
    return roleService.listAll();
}
```

```
@Override
public Map<String, Object> listAll() {
    Map<String,Object> map = new HashMap<>();

    List<Role> roleList = roleDao.findAll();

    logService.save(new Log(Log.SELECT_ACTION, "查询所有角色信息"));

    map.put("rows", roleList);

    return map;
}
```

这是一个查询所有角色信息接口，这段代码的功能是查询数据库中的所有角色信息，并将结果存储在一个`Map<String, Object>`对象中，然后返回该对象。同时，它还保存了一条日志记录，用于记录查询所有角色信息的操作。

接口二十九：

```
@PostMapping("/list")
@RequiresPermissions(value = "角色管理")
public Map<String, Object> list(Integer page, Integer rows, String roleName) {
    return roleService.list(page, rows, roleName);
}
```

```
@Override
public Map<String, Object> list(Integer page, Integer rows, String roleName) {
    Map<String, Object> map = new HashMap<>();
```

```
    int total = roleDao.getRoleCount(roleName);
    page = page == 0 ? 1 : page;
    int offSet = (page - 1) * rows;
    List<Role> roles = roleDao.getRoleList(offSet, rows, roleName);

    logService.save(new Log(Log.SELECT_ACTION, "分页查询角色信息"));

    map.put("total", total);
    map.put("rows", roles);

    return map;
}
```

这是一个分页查询角色信息接口，这段代码的功能是进行分页查询符合条件的角色信息，并将结果存储在一个`Map<String, Object>`对象中，然后返回该对象。同时，它还保存了一条日志记录，用于记录分页查询角色信息的操作。

接口三十：

```
 @RequestMapping("/save")
    @RequiresPermissions(value = "角色管理")
    public ServiceVO save(Role role) {
        return roleService.save(role);
    }
```

```
@Override
public ServiceVO save(Role role) {

    // 角色ID为空时，说明是新增操作，需要先判断角色名是否存在
    if(role.getRoleId() == null) {

        Role exRole = roleDao.findRoleByName(role.getRoleName());

        if(exRole != null) {

            return new ServiceVO<>(ErrorCode.ROLE_EXIST_CODE,
ErrorCode.ROLE_EXIST_MESS);

        }

        logService.save(new Log(Log.INSERT_ACTION, "新增角色:" +
role.getRoleName()));

        roleDao.insertRole(role);

    } else {

        logService.save(new Log(Log.UPDATE_ACTION, "修改角
色:"+role.getRoleName()));

        roleDao.updateRole(role);

    }

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
```

```
}
```

这是一个添加或修改角色信息接口，这段代码的功能是根据传入的角色对象进行保存操作，如果角色ID为空，则执行新增操作，先判断角色名是否已存在。如果角色名已存在，则返回一个带有错误代码和错误消息的`ServiceVO`对象；否则，保存角色信息到数据库，并记录相应的日志。如果角色ID不为空，则执行修改操作，更新角色信息到数据库，并记录相应的日志。最后，无论是新增还是修改操作，都返回一个带有成功代码和成功消息的`ServiceVO`对象作为方法的结果。

接口三十一：

```
@RequestMapping("/delete")
@RequiresPermissions(value = "角色管理")
public ServiceVO delete(Integer roleId) {
    return roleService.delete(roleId);
}
```

```
@Override
public ServiceVO delete(Integer roleId) {
    int count = roleDao.countUserByRoleId(roleId);
    if (count > 0) {
        return new ServiceVO<>(ErrorCode.ROLE_DEL_ERROR_CODE,
ErrorCode.ROLE_DEL_ERROR_MESS);
    }

    roleMenuDao.deleteRoleMenuByRoleId(roleId);

    logService.save(new Log(Log.DELETE_ACTION,"删除角
色:"+roleDao.getRoleById(roleId).getRoleName()));

    roleDao.deleteRole(roleId);

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);

}
```

这是一个删除角色接口，这段代码的功能是根据传入的角色ID进行删除操作。首先，通过统计具有该角色ID的用户数量判断是否能够删除角色，如果存在关联用户则返回一个带有错误代码和错误消息的`ServiceVO`对象。然后，删除角色与菜单之间的关联关系，记录相应的日志，最后删除角色信息，并返回一个带有成功代码和成功消息的`ServiceVO`对象作为方法的结果。

接口三十二：

```
@RequestMapping("/setMenu")
@RequiresPermissions(value = "角色管理")
public ServiceVO setMenu(Integer roleId, String menus) {
    return roleService.setMenu(roleId, menus);
}
```

```
@Override
public ServiceVO setMenu(Integer roleId, String menus) {
    // 先删除当前角色的所有菜单
    roleMenuDao.deleteRoleMenuByRoleId(roleId);
```

```
    // 再赋予当前角色新的菜单
    String[] menuArray = menus.split(",");

    for(String str : menuArray){

        RoleMenu rm = new RoleMenu();

        rm.setRoleId(roleId);

        rm.setMenuId(Integer.parseInt(str));

        roleMenuDao.save(rm);

    }

    logService.save(new Log(Log.UPDATE_ACTION,"设
置"+roleDao.getRoleById(roleId).getRoleName()+"角色的菜单权限"));

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

这是一个设置菜单权限接口，这段代码的作用是根据传入的角色ID和菜单字符串，先删除该角色的所有菜单权限，然后为该角色赋予新的菜单权限，并记录相应的操作日志。最后，返回表示成功的`ServiceVO`对象。

接口三十三：

```
@RequestMapping("/save")
@RequiresPermissions(value = "销售出库")
public ServiceVO save(SaleList saleList, String saleListGoodsStr) {
    return saleListGoodsService.save(saleList, saleListGoodsStr);
}
```

```
@Override
public ServiceVO save(SaleList saleList, String saleListGoodsStr) {
    // 使用谷歌Gson将JSON字符串数组转换成具体的集合
    Gson gson = new Gson();

    List<SaleListGoods> saleListGoodsList = gson.fromJson(saleListGoodsStr, new
TypeToken<List<SaleListGoods>>() {
    }.getType());

    // 设置当前操作用户
    User currentUser = userDao.findUserByName((String)
SecurityUtils.getSubject().getPrincipal());

    saleList.setUserId(currentUser.getUserId());

    // 保存销售单信息
    saleListGoodsDao.saveSaleList(saleList);

    // 保存销售单商品信息
    for (SaleListGoods saleListGoods : saleListGoodsList) {

        saleListGoods.setSaleListId(saleList.getSaleListId());
```

```
        saleListGoodsDao.saveSaleListGoods(saleListGoods);

        // 修改商品库存，状态
        Goods goods = goodsDao.findByGoodsId(saleListGoods.getGoodsId());

        goods.setInventoryQuantity(goods.getInventoryQuantity() -
saleListGoods.getGoodsNum());

        goods.setState(2);

        goodsDao.updateGoods(goods);

    }

    // 保存日志
    logService.save(new Log(Log.INSERT_ACTION, "新增销售单：" +
saleList.getSaleNumber()));

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

这是一个保存销售单信息接口，这段代码的作用是将传入的销售单信息和销售单商品信息保存到数据库中，并进行相应的处理，如更新商品库存数量和状态，记录操作日志。最后，返回表示成功的`ServiceVO`对象。

接口三十四：

```
@RequestMapping("/list")
@RequiresPermissions(value={"销售单据查询","客户统计"},logical= Logical.OR)
public Map<String,Object> list(String saleNumber, Integer customerId, Integer
state, String sTime,
                               String eTime) {
    return saleListGoodsService.list(saleNumber, customerId, state, sTime,
eTime);
}
```

```
@Override
public Map<String, Object> list(String saleNumber, Integer customerId, Integer
state, String sTime, String eTime) {
    Map<String, Object> result = new HashMap<>();

    try {

        List<SaleList> saleListList = saleListGoodsDao.getSalelist(saleNumber,
customerId, state, sTime, eTime);

        logService.save(new Log(Log.SELECT_ACTION, "销售单据查询"));

        result.put("rows", saleListList);

    } catch (Exception e) {

        e.printStackTrace();

    }
    return result;
```

```
}
```

这是一个查询销售单接口，这段代码的作用是根据传入的条件查询销售单据列表，并将查询结果存储在一个`HashMap`对象中，然后返回该对象作为方法的结果。在查询过程中，还会记录操作日志。

接口三十五：

```
@RequestMapping("/goodsList")
@RequiresPermissions(value={"销售单据查询","客户统计"},logical=Logical.OR)
public Map<String,Object> goodsList(Integer saleListId) {
    return saleListGoodsService.goodsList(saleListId);
}
```

```
@Override
public Map<String, Object> goodsList(Integer saleListId) {
    Map<String, Object> map = new HashMap<>();

    try {

        List<SaleListGoods> saleListGoodsList =
saleListGoodsDao.getSaleListGoodsBySaleListId(saleListId);

        logService.save(new Log(Log.SELECT_ACTION, "销售单商品信息查询"));

        map.put("rows", saleListGoodsList);

    } catch (Exception e) {

        e.printStackTrace();

    }

    return map;
}
```

这是一个查询销售单商品信息接口，这段代码的作用是根据传入的销售单ID查询对应的销售单商品信息列表，并将查询结果存储在一个`HashMap`对象中，然后返回该对象作为方法的结果。在查询过程中，还会记录操作日志。

接口三十六：

```
@RequestMapping("/delete")
@RequiresPermissions(value = "销售单据查询")
public ServiceVO delete(Integer saleListId) {
    return saleListGoodsService.delete(saleListId);
}
```

```
@Override
public ServiceVO delete(Integer saleListId) {

    logService.save(new Log(Log.DELETE_ACTION, "删除销售单：" +
saleListGoodsDao.getSaleList(saleListId).getSaleNumber()));

    saleListGoodsDao.deleteSaleListGoodsBySaleListId(saleListId);

    saleListGoodsDao.deleteSaleListById(saleListId);

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

这是一个删除销售单及商品信息接口，这段代码的作用是根据传入的销售单ID，先删除该销售单ID对应的销售单商品信息，然后再删除销售单信息，并记录相应的操作日志。最后，返回表示成功的`ServiceVO`对象。

接口三十七：

```
@RequestMapping("/updateState")
@RequiresPermissions(value = "供应商统计")
public ServiceVO updateState(Integer saleListId) {
    return saleListGoodsService.updateState(saleListId);
}
```

```
@Override
public ServiceVO updateState(Integer saleListId) {

    saleListGoodsDao.updateState(saleListId);

    logService.save(new Log(Log.DELETE_ACTION, "支付结算销售单：" +
saleListGoodsDao.getSaleList(saleListId).getSaleNumber()));

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);

}
```

这是一个修改销售单付款状态接口，这段代码的作用是根据传入的销售单ID，先删除该销售单ID对应的销售单商品信息，然后再删除销售单信息，并记录相应的操作日志。最后，返回表示成功的`ServiceVO`对象。

接口三十八：

```
@RequestMapping("/count")
@RequiresPermissions(value="商品销售统计")
public String count(String sTime, String eTime ,Integer goodsTypeId, String
codeOrName) {
    return saleListGoodsService.count(sTime, eTime, goodsTypeId, codeOrName);
}
```

```
@Override
public String count(String sTime, String eTime, Integer goodsTypeId, String
codeOrName) {
```

```
    JsonArray result = new JsonArray();

    try {

        List<SaleList> saleListList = saleListGoodsDao.getSalelist(null, null,
null, sTime, eTime);

        for (SaleList pl : saleListList) {

            List<SaleListGoods> saleListGoodsList = saleListGoodsDao
                    .getSaleListGoods(pl.getSaleListId(), goodsTypeId,
codeOrName);

            for (SaleListGoods pg : saleListGoodsList) {

                JsonObject obj = new JsonObject();

                obj.addProperty("number", pl.getSaleNumber());

                obj.addProperty("date", pl.getSaleDate());

                obj.addProperty("customerName", pl.getCustomerName());

                obj.addProperty("code", pg.getGoodsCode());

                obj.addProperty("name", pg.getGoodsName());

                obj.addProperty("model", pg.getGoodsModel());

                obj.addProperty("goodsType",
goodsTypeDao.getGoodsTypeById(pg.getGoodsTypeId()).getGoodsTypeName());

                obj.addProperty("unit", pg.getGoodsUnit());

                obj.addProperty("price", pg.getPrice());

                obj.addProperty("num", pg.getGoodsNum());

                obj.addProperty("total", pg.getTotal());

                result.add(obj);

            }
        }

        logService.save(new Log(Log.SELECT_ACTION, "销售商品统计查询"));

    } catch (Exception e) {

        e.printStackTrace();

    }

    return result.toString();
}
```

这是一个销售商品统计接口，这段代码的功能是根据给定的时间范围、商品类型和商品编码/名称统计销售商品，并将结果以JSON格式返回。

接口三十九：

```
@RequestMapping("/getSaleDataByDay")
@RequiresPermissions(value = "按日统计分析")
public String getSaleDataByDay(String sTime, String eTime) {
    return saleListGoodsService.getSaleDataByDay(sTime, eTime);
}
```

```
@Override
public String getSaleDataByDay(String sTime, String eTime) {
    JsonArray result = new JsonArray();

    try {

        // 获取所有的时间段日期
        List<String> dateList = DateUtil.getTimeSlotByDay(sTime, eTime);

        // 查询按日统计的数据
        List<SaleData> obList = saleListGoodsDao.getSaleDataByDay(sTime, eTime);

        // 按日统计的数据，如果该日期没有数据的话，则不会有显示，我们的需求是，如果该日期没有销售数据的话，也应该显示为0，所以需要进行特殊处理
        for (String date : dateList) {

            JsonObject obj = new JsonObject();

            boolean flag = false;

            for (SaleData o : obList) {

                if (o.getDate().equals(date)) {

                    obj.addProperty("date", o.getDate()); //日期

                    obj.addProperty("saleTotal", BigDecimalUtil.keepTwoDecimalPlaces(o.getSaleTotal())); //销售总额

                    obj.addProperty("purchasingTotal", BigDecimalUtil.keepTwoDecimalPlaces(o.getPurchasingTotal())); //成本总额

                    obj.addProperty("profit", BigDecimalUtil.keepTwoDecimalPlaces(o.getSaleTotal() - o.getPurchasingTotal())); //利润

                    flag = true;

                    break;

                }

            }

            if (!flag) {// 如果没有销售数据，那么也需要设置该日的销售数据默认为0
```

```
                obj.addProperty("date", date); //日期

                obj.addProperty("saleTotal", 0); //销售总额

                obj.addProperty("purchasingTotal", 0); //成本总额

                obj.addProperty("profit", 0); //利润

            }

            result.add(obj);

        }

        logService.save(new Log(Log.SELECT_ACTION, "查询按日统计分析数据"));

    } catch (Exception e) {

        e.printStackTrace();

    }

    return result.toString();
}
```

这是一个按日统计销售情况接口，这段代码的作用是根据传入的起始时间和结束时间，查询该时间范围内按日统计的销售数据。对于每个日期，如果该日期没有销售数据，也会将该日期的销售数据设置为0，并将统计结果以JSON字符串形式返回。在查询过程中，还会记录相应的操作日志。

接口四十：

```
@RequestMapping("/getSaleDataByMonth")
@RequiresPermissions(value = "按月统计分析")
public String getSaleDataByMonth(String sTime, String eTime) {
    return saleListGoodsService.getSaleDataByMonth(sTime, eTime);
}
```

```
@Override
public String getSaleDataByMonth(String sTime, String eTime) {
    JsonArray result = new JsonArray();

    try {

        // 获取所有的时间段日期
        List<String> dateList = DateUtil.getTimeSlotByMonth(sTime, eTime);

        // 查询按日统计的数据
        List<SaleData> obList = saleListGoodsDao.getSaleDataByMonth(sTime,
eTime);

        // 按日统计的数据，如果该日期没有数据的话，则不会有显示，我们的需求是，如果该日期没有销售数据的话，也应该显示为0，所以需要进行特殊处理
        for (String date : dateList) {

            JsonObject obj = new JsonObject();
```

```
            boolean flag = false;

            for (SaleData o : obList) {

                if (o.getDate().equals(date)) {

                    obj.addProperty("date", o.getDate()); //日期

                    obj.addProperty("saleTotal",
BigDecimalUtil.keepTwoDecimalPlaces(o.getSaleTotal())); //销售总额

                    obj.addProperty("purchasingTotal",
BigDecimalUtil.keepTwoDecimalPlaces(o.getPurchasingTotal())); //成本总额

                    obj.addProperty("profit",
BigDecimalUtil.keepTwoDecimalPlaces(o.getSaleTotal() - o.getPurchasingTotal()));
//利润

                    flag = true;

                }

            }

            if (!flag) {// 如果没有销售数据，那么也需要设置该日的销售数据默认为0

                obj.addProperty("date", date); //日期

                obj.addProperty("saleTotal", 0); //销售总额

                obj.addProperty("purchasingTotal", 0); //成本总额

                obj.addProperty("profit", 0); //利润

            }

            result.add(obj);

        }

        logService.save(new Log(Log.SELECT_ACTION, "查询按月统计分析数据"));

    } catch (Exception e) {

        e.printStackTrace();

    }

    return result.toString();
}
```

这是一个按月统计销售情况接口，这段代码的功能是按月统计销售数据，并返回每天的销售总额、成本总额和利润。如果某个日期没有对应的销售数据，则默认将其销售数据设置为0。返回的结果以JSON数组的形式表示。

接口四十一:

```
@RequestMapping("/getSaleDataByYear")
@RequiresPermissions(value = "按年统计分析")
public String getSaleDataByYear(String sTime, String eTime) {
    return saleListGoodsService.getSaleDataByYearForCustomer(sTime, eTime);
}
```

```
@Override
public String getSaleDataByYearForCustomer(String sTime, String eTime) {
    JsonArray result = new JsonArray();

    try {

        // 查询按日统计的数据
        List<SaleDataCustomer> obList =
saleListGoodsDao.getSaleDataByYearForCustomer(sTime, eTime);


        for (SaleDataCustomer o : obList) {
            JsonObject obj = new JsonObject();

            obj.addProperty("value",
BigDecimalUtil.keepTwoDecimalPlaces(o.getSaleTotal_Customer())); //销售总额
            obj.addProperty("name", o.getCustomer_name()); //成本总额

            result.add(obj);
        }

        logService.save(new Log(Log.SELECT_ACTION, "查询按年统计分析数据"));

    } catch (Exception e) {

        e.printStackTrace();

    }

    return result.toString();
}
```

这是一个按年统计销售情况接口，这段代码的作用是根据传入的起始时间和结束时间，查询该时间范围内按年统计的销售数据，按客户进行统计。将每个客户的销售总额和客户名称作为键值对存储在`JsonObject`对象中，并将统计结果以JSON字符串形式返回。在查询过程中，还会记录相应的操作日志。

接口四十二:

```
@RequestMapping("/getSaleDataByYearGoods")
@RequiresPermissions(value = "按年统计分析")
public String getSaleDataByYearGoods(String sTime, String eTime) {
    return saleListGoodsService.getSaleDataByYearForGoods(sTime, eTime);
}
```

```
@Override
```

```
public String getSaleDataByYearForGoods(String sTime, String eTime) {
    JsonArray result = new JsonArray();

    try {

        // 查询按日统计的数据
        List<SaleDataCustomer> obList =
saleListGoodsDao.getSaleDataByYearForGoods(sTime, eTime);


        for (SaleDataCustomer o : obList) {
            JsonObject obj = new JsonObject();

            obj.addProperty("value",
BigDecimalUtil.keepTwoDecimalPlaces(o.getSaleTotal_Customer())); //销售总额
            obj.addProperty("name", o.getCustomer_name()); //成本总额

            result.add(obj);
        }

        logService.save(new Log(Log.SELECT_ACTION, "查询按年统计分析数据"));

    } catch (Exception e) {

        e.printStackTrace();

    }

    return result.toString();
}
```

这段代码的作用是根据传入的起始时间和结束时间，查询该时间范围内按年统计的销售数据，按商品进行统计。将每个商品的销售总额和商品名称作为键值对存储在`JsonObject`对象中，并将统计结果以JSON字符串形式返回。在查询过程中，还会记录相应的操作日志。

接口四十三：

```
@RequestMapping("/getSaleDataByYearSupplier")
@RequiresPermissions(value = "按年统计分析")
public String getSaleDataByYearSupplier(String sTime, String eTime) {
    return saleListGoodsService.getSaleDataByYearForSupplier(sTime, eTime);
}
```

```
@Override
public String getSaleDataByYearForSupplier(String sTime, String eTime) {


    List<List<Object>> data = new ArrayList<>();

    List<Object> header = new ArrayList<>();
    header.add("supplier");
    header.add("year");
    header.add("profit");
    data.add(header);
```

```
    List<SaleDataSupplier> obList =
saleListGoodsDao.getSaleDataByYearForSupplier(sTime, eTime);

    for(SaleDataSupplier o:obList){

        List<Object> record = new ArrayList<>();
        record.add(o.getSupplier());
        record.add(o.getDate());
        record.add(BigDecimalUtil.keepTwoDecimalPlaces(o.getSaleTotal() -
o.getPurchasingTotal()));

        data.add(record);

    }

    Gson gson = new Gson();
    String json = gson.toJson(data);

    return json;
}
```

这段代码的功能是按年统计供应商的销售数据，并将结果以JSON格式返回。返回的JSON字符串包括"supplier"（供应商）、"year"（年份）和"profit"（利润）三个列名作为表头，后续行则是每个供应商对应的年份和利润数据。

接口四十四：

```
@RequestMapping("/getSaleDataByMonthForUserSale")
@RequiresPermissions(value = "按销售人员统计分析")
public String getSaleDataByMonthForUserSale(String ID) {
    return saleListGoodsService.getSaleDataByMonthForUserSale(ID);
}
```

```
@Override
public String getSaleDataByMonthForUserSale(String ID) {
    JsonArray result = new JsonArray();

    try {

        // 获取所有的时间段日期
        List<String> dateList = DateUtil.getTimeSlotByMonth("2022-07-01","2023-
07-31");

        // 查询按日统计的数据
        List<SaleUser> obList =
saleListGoodsDao.getSaleDataByMonthForUserSale(ID);

        // 按日统计的数据，如果该日期没有数据的话，则不会有显示，我们的需求是，如果该日期没有销售数据的话，也应该显示为0，所以需要进行特殊处理
        for (String date : dateList) {

            JsonObject obj = new JsonObject();

            boolean flag = false;

            for (SaleUser o : obList) {
```

```
                if (o.getDate().equals(date)) {
                    obj.addProperty("date", o.getDate()); //日期
                    obj.addProperty("saleTotal",
BigDecimalUtil.keepTwoDecimalPlaces(o.getSaleTotal())); //销售总额

                    flag = true;
                }
            }
            if (!flag) {// 如果没有销售数据，那么也需要设置该日的销售数据默认为0
                obj.addProperty("date", date); //日期
                obj.addProperty("saleTotal", 0); //销售总额

            }
            result.add(obj);
        }
        logService.save(new Log(Log.SELECT_ACTION, "查询销售人员分析数据"));
    } catch (Exception e) {
        e.printStackTrace();
    }
    return result.toString();
}
```

这段代码的作用是根据传入的销售人员ID，查询该销售人员在指定时间范围内按月统计的销售数据。对于每个月份，如果该月份没有销售数据，也会将该月份的销售数据设置为0，并将统计结果以JSON字符串形式返回。在查询过程中，还会记录相应的操作日志。

接口四十五:

```
@GetMapping("/IDlist")
@RequiresPermissions(value = "按销售人员统计分析")
public String IDlist() {
    return saleListGoodsService.getUserIDlist();
}
```

```
@Override
public String getUserIDlist() {
    JsonArray result = new JsonArray();
    List<UserSale> obList = saleListGoodsDao.getUserIDList();
```

```
    try {
        for (UserSale user : obList) {

            JsonObject obj = new JsonObject();

            obj.addProperty("ID", user.getUserdeid());

            result.add(obj);
        }

        logService.save(new Log(Log.SELECT_ACTION, "查询用户ID数据"));

    } catch (Exception e) {

        e.printStackTrace();

    }

    return result.toString();
}
```

这段代码的功能是查询用户ID列表，并将每个用户ID封装成一个`JsonObject`对象，然后将所有的对象添加到一个`JsonArray`中。最终，将结果以JSON字符串的形式返回。

接口四十六：

```
@RequestMapping("/getComboboxList")
@RequiresPermissions(value={"进货入库","退货出库","进货单据查询","退货单据查询","供应商统计"},logical= Logical.OR)
public List<Supplier> getComboboxList(String q) {
    return supplierService.getComboboxList(q);
}
```

```
@Override
public List<Supplier> getComboboxList(String q) {
    return supplierDao.getSupplierListByNameLike(q);
}
```

这是一个查询下拉框供应商信息接口，这段代码的功能是根据给定的关键字 `q` 进行模糊查询，获取匹配供应商名称的列表，并将列表作为结果返回。通常用于提供给前端页面的下拉框（combobox）数据源，以便用户选择合适的供应商名称。

接口四十七：

```
@RequestMapping("/list")
@RequiresPermissions(value = "供应商管理")// 有供应商管理菜单权限的才给予调用
public Map<String,Object> list(Integer page, Integer rows, String supplierName)
{
    return supplierService.list(page, rows, supplierName);
}
```

```
@Override
public Map<String, Object> list(Integer page, Integer rows, String supplierName)
{
```

```
    Map<String,Object> map = new HashMap<>();

    page = page == 0 ? 1 : page;
    int offSet = (page - 1) * rows;

    List<Supplier> suppliers = supplierDao.getSupplierList(offSet, rows,
supplierName);

    logService.save(new Log(Log.SELECT_ACTION,"分页查询供应商"));

    map.put("total", supplierDao.getSupplierCount(supplierName));

    map.put("rows", suppliers);

    return map;
}
```

这是一个分页查询供应商接口，这段代码的作用是进行分页查询供应商信息。根据传入的页码、每页显示的行数和供应商名称，调用`supplierDao`对象的方法进行查询，将查询结果封装到`map`对象中并返回。在查询过程中，还会记录相应的操作日志。

接口四十八：

```
@RequestMapping("/save")
@RequiresPermissions(value = "供应商管理")
public ServiceVO save(Supplier supplier) {
    return supplierService.save(supplier);
}
```

```
@Override
public ServiceVO save(Supplier supplier) {

    if(supplier.getSupplierId() == null){

        supplierDao.saveSupplier(supplier);
        logService.save(new Log(Log.INSERT_ACTION,"添加供应
商:"+supplier.getSupplierName()));

    }else{

        supplierDao.updateSupplier(supplier);
        logService.save(new Log(Log.UPDATE_ACTION,"修改供应
商:"+supplier.getSupplierName()));

    }

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);

}
```

这是一个添加或修改供应商接口，这段代码的功能是根据传入的`Supplier`对象执行添加或修改操作，并在数据库中保存相应的数据。同时，根据操作类型记录相应的日志信息。方法返回的`ServiceVO`对象用于表示操作结果和相关消息。

接口四十九：

```
@RequestMapping("/delete")
@RequiresPermissions(value = "供应商管理")
public ServiceVO delete(String ids) {
    return supplierService.delete(ids);
}
```

```
@Override
public ServiceVO delete(String ids) {

    String[] idArray = ids.split(",");

    for (String id : idArray) {

        logService.save(new Log(Log.DELETE_ACTION,
                "删除供应商:" +
supplierDao.getSupplierById(Integer.parseInt(id)).getSupplierName()));

        supplierDao.deleteSupplier(Integer.parseInt(id));

    }
    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

这是一个删除供应商接口，这段代码的作用是根据传入的供应商ID字符串，将其分隔为供应商ID数组，并逐个删除对应的供应商信息。在删除每个供应商之前，还会记录相应的操作日志。最后返回一个表示成功操作的`ServiceVO`对象。

接口五十：

```
@RequestMapping("/save")
@ResponseBody
@RequiresPermissions(value = "商品管理")
public ServiceVO save(Unit unit) {
    return unitService.save(unit);
}
```

```
@Override
public ServiceVO save(Unit unit) {

    logService.save(new Log(Log.INSERT_ACTION,"添加商品单位:"+unit.getUnitName()));

    unitDao.saveUnit(unit);

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

这是添加或修改商品单位接口，这段代码的作用是将商品单位信息保存到数据库，并在保存完成后记录相应的操作日志。最后返回一个表示成功操作的`ServiceVO`对象。

接口五十一：

```
@RequestMapping("/delete")
@ResponseBody
@RequiresPermissions(value = "商品管理")
public ServiceVO delete(Integer unitId) {
    return unitService.delete(unitId);
}
```

```
@Override
public ServiceVO delete(Integer unitId) {

        logService.save(new Log(Log.DELETE_ACTION,"删除商品单
位:"+unitDao.getUnitByUnitId(unitId).getUnitName()));

        unitDao.delete(unitId);

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

这是删除商品单位接口，这段代码的功能是根据传入的 `unitId` 执行删除操作，即从数据库中删除对应的商品单位信息。同时，根据删除的单位名称记录相应的日志信息。方法返回的 `ServiceVO` 对象用于表示操作结果和相关消息。

接口五十二：

```
@RequestMapping("/list")
@ResponseBody
@RequiresPermissions(value = "商品管理")
public Map<String,Object> list() {
    return unitService.list();
}
```

```
@Override
public Map<String, Object> list() {
    Map<String,Object> map = new HashMap<>();

    List<Unit> unitList = unitDao.listAll();

    map.put("rows", unitList);

    return map;
}
```

这是查询所有商品单位接口，这段代码的作用是查询所有的商品单位信息，并将查询结果封装到 `map` 对象中并返回。在查询过程中，没有进行特殊处理或记录操作日志。

接口五十三：

```
@PostMapping("/login")
@ResponseBody
public ServiceVO login(@RequestBody UserLogin userLogin, HttpSession session){
    return userService.login(userLogin, session);
}
```

```
@Override
public ServiceVO login(UserLogin userLogin, HttpSession session) {
    try {
        // 校验图片验证码是否正确

if(!userLogin.getImageCode().toUpperCase().equals(session.getAttribute("checkcod
e"))){
            return new ServiceVO(ErrorCode.VERIFY_CODE_ERROR_CODE,
ErrorCode.VERIFY_CODE_ERROR_MESS);
        }

        //开始进行登录校验
        Subject subject = SecurityUtils.getSubject();

        UsernamePasswordToken token = new
UsernamePasswordToken(userLogin.getUserName(), userLogin.getPassword());

        subject.login(token);

        // 登录成功后，开始查询用户的角色
        User user = userDao.findUserByName(userLogin.getUserName());

        List<Role> roles = roleDao.getRoleByUserId(user.getUserId());

        session.setAttribute("currentUser", user);

        logService.save(new Log(Log.LOGIN_ACTION, "登录系统"));

        return new ServiceVO<>(SuccessCode.SUCCESS_CODE,
SuccessCode.SUCCESS_MESS, roles);

    } catch (AuthenticationException e) {// 如果抛出AuthenticationException异常，说
明登录校验未通过
        e.printStackTrace();
        return new ServiceVO(ErrorCode.NAME_PASSWORD_ERROR_CODE,
ErrorCode.NAME_PASSWORD_ERROR_MESS);
    } catch(Exception e){
        e.printStackTrace();
        return new ServiceVO(ErrorCode.REQ_ERROR_CODE,
ErrorCode.REQ_ERROR_MESS);
    }
}
```

这是系统登录接口，这段代码的功能是进行用户登录验证。首先校验图片验证码是否正确，然后使用提供的用户名和密码创建身份验证令牌，通过调用 Subject 的 login 方法进行身份验证。如果验证通过，查询用户的角色信息，并将当前用户对象存储在会话中。同时，记录登录日志。最后，根据不同的情况返回相应的结果。

接口五十四：

```
@GetMapping("/loadUserInfo")
@ResponseBody
public Map<String,Object> loadUserInfo(HttpSession session) {
    return userService.loadUserInfo(session);
}
```

```
@Override
public Map<String, Object> loadUserInfo(HttpSession session) {
    Map<String,Object> map = new HashMap<>();

    User user = (User)session.getAttribute("currentUser");
    Role role = (Role) session.getAttribute("currentRole");

    map.put("userName", user.getTrueName());
    map.put("roleName", role.getRoleName());

    return map;
}
```

这是一个从缓存中获取当前登录的用户相关信息，包括用户真实姓名和角色名称接口，这段代码的作用是从`HttpSession`中获取当前用户的信息，包括用户名和角色名称，并将其封装到`map`对象中并返回。这样可以方便地在其他地方获取当前用户的相关信息。

接口五十五：

```
@RequestMapping("/list")
@ResponseBody
@RequiresPermissions(value = "用户管理")// 有用户管理菜单权限的才给予调用
public Map<String, Object> list(Integer page,Integer rows,String userName) {
    return userService.list(page, rows, userName);
}
```

```
@Override
public Map<String, Object> list(Integer page, Integer rows, String userName) {
    Map<String,Object> map = new HashMap<>();

    page = page == 0 ? 1 : page;
    int offSet = (page - 1) * rows;
    List<User> users = userDao.getUserList(offSet, rows, userName);

    for (User user : users) {

        List<Role> roles = roleDao.getRoleByUserId(user.getUserId());

        StringBuffer sb = new StringBuffer();

        for(Role role : roles) {

            sb.append(","+role.getRoleName());

        }
```

```
        user.setRoles(sb.toString().replaceFirst(",", ""));

    }

    logService.save(new Log(Log.SELECT_ACTION,"分页查询用户信息"));

    map.put("total", userDao.getUserCount(userName));

    map.put("rows", users);

    return map;
}
```

这是一个分页查询用户信息接口，这段代码的功能是进行分页查询用户信息。根据传入的页数、每页记录数和用户名，在数据库中查询符合条件的用户列表。对于每个用户，还会获取其角色信息，并将角色名称拼接成一个字符串，并设置回用户对象中。最后，记录查询日志，并将总记录数和查询到的用户列表作为结果返回。

接口五十六:

```
@RequestMapping(value = "/save")
@ResponseBody
@RequiresPermissions(value = "用户管理")
public ServiceVO save(User user) {
    return userService.save(user);
}
```

```
@Override
public ServiceVO save(User user) {

    // 用户ID为空时，说明是新增操作，需要先判断用户名是否存在
    if(user.getUserId() == null){

        User exUser = userDao.findUserByName(user.getUserName());

        if (exUser != null) {
            return new ServiceVO(ErrorCode.ACCOUNT_EXIST_CODE,
ErrorCode.ACCOUNT_EXIST_MESS);
        }

        userDao.addUser(user);
        logService.save(new Log(Log.INSERT_ACTION,"添加用
户:"+user.getUserName()));

    } else {

        userDao.updateUser(user);
        logService.save(new Log(Log.UPDATE_ACTION,"修改用
户:"+user.getUserName()));

    }

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

这是一个添加或修改用户信息接口，这段代码的作用是根据传入的用户对象，判断是新增操作还是更新操作。如果是新增操作，会先判断用户名是否已存在，然后将用户信息添加到数据库中，并记录相应的操作日志。如果是更新操作，直接将用户信息更新到数据库中，并记录相应的操作日志。最后返回一个表示操作结果的`ServiceVO`对象。

接口五十七：

```
@RequestMapping("/setRole")
@ResponseBody
@RequiresPermissions(value = "用户管理")
public ServiceVO setRole(Integer userId, String roles) {
    return userService.setRole(userId, roles);
}
```

```
@Override
public ServiceVO setRole(Integer userId, String roles) {

    // 先删除当前用户的所有角色
    userRoleDao.deleteUserRoleByUserId(userId);

    // 再赋予当前用户新的角色
    String[] roleArray = roles.split(",");

    for(String str : roleArray){

        UserRole ur = new UserRole();

        ur.setRoleId(Integer.parseInt(str));

        ur.setUserId(userId);

        userRoleDao.addUserRole(ur);

    }

    logService.save(new Log(Log.UPDATE_ACTION,"设置用
户"+userDao.getUserById(userId).getUserName()+"的角色权限"));

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

这是一个设置用户角色这段代码的功能是为指定用户设置角色权限。首先，删除当前用户的所有角色信息。然后，根据传入的角色字符串，逐个创建`UserRole`对象，并赋予当前用户新的角色。最后，记录设置角色权限的日志，并返回一个包含成功状态码和成功消息的`ServiceVO`对象。

接口五十八：

```
@RequestMapping("/updatePassword")
@ResponseBody
@RequiresPermissions(value = "修改密码")
public ServiceVO updatePassword(String newPassword, HttpSession session) {
    return userService.updatePassword(newPassword, session);
}
```

```
@Override
public ServiceVO updatePassword(String newPassword, HttpSession session) {
    User user = (User) session.getAttribute("currentUser");

    user.setPassword(newPassword);

    userDao.updateUser(user);

    logService.save(new Log(Log.UPDATE_ACTION,"修改密码"));

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

这是一个修改密码接口，这段代码的作用是根据传入的新密码，更新当前用户的密码。在更新密码之后，会记录相应的操作日志。最后返回一个表示操作结果的`ServiceVO`对象。

接口五十九：

```
@RequestMapping("/delete")
@ResponseBody
@RequiresPermissions(value = "用户管理")
public ServiceVO delete(Integer userId) {
    return userService.delete(userId);
}
```

```
@Override
public ServiceVO delete(Integer userId) {

    logService.save(new Log(Log.DELETE_ACTION,"删除用
户:"+userDao.getUserById(userId).getUserName()));

    userRoleDao.deleteUserRoleByUserId(userId);

    userDao.deleteUser(userId);

    return new ServiceVO<>(SuccessCode.SUCCESS_CODE, SuccessCode.SUCCESS_MESS);
}
```

这是一个删除用户信息接口，这段代码的功能是删除指定用户。首先，记录删除用户的日志。然后，删除该用户的所有角色信息。最后，删除该用户。最终返回一个包含成功状态码和成功消息的`ServiceVO`对象。

接口六十：

```
@GetMapping("/logOut")
@RequiresPermissions(value = "安全退出")
public String logOut() {

    logService.save(new Log(Log.LOGOUT_ACTION,"用户注销"));

    //清除shiro用户信息
    SecurityUtils.getSubject().logout();

    return "redirect:login.html";
}
```

```
public void logout() {
    try {
        this.clearRunAsIdentitiesInternal();
        this.securityManager.logout(this);
    } finally {
        this.session = null;
        this.principals = null;
        this.authenticated = false;
    }

}
```

这是一个安全退出接口，这段代码的作用是执行用户注销操作，包括清除身份、会话信息以及标记为未认证状态。但由于代码片段不完整，无法提供更具体的解释和执行流程。完整的代码上下文可能包含更多与用户注销相关的逻辑和处理。