**[分页实现](http://doc.ruoyi.vip/" \l "/standard/htsc?id=%e5%88%86%e9%a1%b5%e5%ae%9e%e7%8e%b0)**

前端基于Bootstrap的轻量级表格插件 Bootstrap Table  
后端分页组件使用Mybatis分页插件 PageHelper

**分页实现流程**

1、前端调用封装好的方法$.table.init，传入后台url。

var options = {

url: prefix + "/list",

columns: [{

field: 'id',

title: '主键'

},

{

field: 'name',

title: '名称'

}]

};

$.table.init(options);

2、后台实现查询逻辑，调用startPage()方法即可自动完成服务端分页。

@PostMapping("/list")

@ResponseBody

public TableDataInfo list(User user)

{

startPage(); // 此方法配合前端完成自动分页

List<User> list = userService.selectUserList(user);

return getDataTable(list);

}

注意：启动分页关键代码startPage()（只对该语句以后的第一个查询语句得到的数据进行分页）  
如果改为其他数据库需修改配置application.yml helperDialect=你的数据库

[**导入导出**](http://doc.ruoyi.vip/#/standard/htsc?id=%e5%af%bc%e5%85%a5%e5%af%bc%e5%87%ba)

导入导出使用 Apache POI，目前支持参数如下

| **参数** | **类型** | **默认值** | **描述** |
| --- | --- | --- | --- |
| name | String | 空 | 导出到Excel中的名字 |
| dateFormat | String | 空 | 日期格式, 如: yyyy-MM-dd |
| readConverterExp | String | 空 | 读取内容转表达式 (如: 0=男,1=女,2=未知) |
| height | String | 14 | 导出时在excel中每个列的高度 单位为字符 |
| width | String | 16 | 导出时在excel中每个列的宽 单位为字符 |
| suffix | String | 空 | 文字后缀,如% 90 变成90% |
| defaultValue | String | 空 | 当值为空时,字段的默认值 |
| prompt | String | 空 | 提示信息 |
| combo | String | Null | 设置只能选择不能输入的列内容 |
| isExport | String | true | 是否导出数据,应对需求:有时我们需要导出一份模板,这是标题需要但内容需要用户手工填写 |
| targetAttr | String | 空 | 另一个类中的属性名称,支持多级获取,以小数点隔开 |
| type | Enum | Type.ALL | 字段类型（0：导出导入；1：仅导出；2：仅导入） |

**导出实现流程**

1、前端调用封装好的方法$.table.init，传入后台exportUrl。

var options = {

exportUrl: prefix + "/export",

columns: [{

field: 'id',

title: '主键'

},

{

field: 'name',

title: '名称'

}]

};

$.table.init(options);

2、在实体变量上添加@Excel注解。

@Excel(name = "用户序号")

private Long id;

@Excel(name = "用户名称")

private String userName;

3、在Controller添加导出方法

@PostMapping("/export")

@ResponseBody

public AjaxResult export(User user)

{

List<User> list = userService.selectUserList(user);

ExcelUtil<User> util = new ExcelUtil<User>(User.class);

return util.exportExcel(list, "用户数据");

}

**导入实现流程**

1、前端调用封装好的方法$.table.init，传入后台importUrl。

var options = {

importUrl: prefix + "/importData",

columns: [{

field: 'id',

title: '主键'

},

{

field: 'name',

title: '名称'

}]

};

$.table.init(options);

2、在实体变量上添加@Excel注解，默认为导出导入，也可以单独设置仅导入Type.IMPORT

@Excel(name = "用户序号")

private Long id;

@Excel(name = "部门编号", type = Type.IMPORT)

private Long deptId;

@Excel(name = "用户名称")

private String userName;

3、在Controller添加导入方法，updateSupport属性为是否存在则覆盖（可选）

@PostMapping("/importData")

@ResponseBody

public AjaxResult importData(MultipartFile file, boolean updateSupport) throws Exception

{

ExcelUtil<SysUser> util = new ExcelUtil<SysUser>(SysUser.class);

List<SysUser> userList = util.importExcel(file.getInputStream());

String operName = ShiroUtils.getSysUser().getLoginName();

String message = userService.importUser(userList, updateSupport, operName);

return AjaxResult.success(message);

}

[**上传下载**](http://doc.ruoyi.vip/#/standard/htsc?id=%e4%b8%8a%e4%bc%a0%e4%b8%8b%e8%bd%bd)

首先创建一张上传文件的表，例如：

drop table if exists sys\_file;

create table sys\_file (

fileid int(11) not null auto\_increment comment '文件id',

filename varchar(50) default '' comment '文件名称',

filepath varchar(255) default '' comment '文件路径',

primary key (fileid)

) engine=innodb auto\_increment=200 default charset=utf8 comment = '文件表';

**上传实现流程**

1、参考示例代码。

function submitHandler() {

if ($.validate.form()) {

uploadFile();

}

}

function uploadFile() {

var formData = new FormData();

if($('#file')[0].files[0] == null) {

$.modal.alertWarning("请先选择文件路径");

return false;

}

formData.append('fileName', $("#fileName").val());

formData.append('file', $('#file')[0].files[0]);

$.ajax({

url: prefix + "/add",

type: 'post',

cache: false,

data: formData,

processData: false,

contentType: false,

dataType: "json",

success: function(result) {

$.operate.successCallback(result);

}

});

}

2、在Controller添加对应上传方法

@Autowired

private ServerConfig serverConfig;

@PostMapping("/add")

@ResponseBody

public AjaxResult addSave(MultipartFile file, SysFile sysFile) throws IOException

{

// 上传文件路径

String filePath = Global.getUploadPath();

// 上传并返回新文件名称

String fileName = FileUploadUtils.upload(filePath, file);

sysFile.setFilePath(fileName);

return toAjax(sysFileService.insertSysFile(sysFile));

}

3、上传成功后需要预览可以对该属性格式化处理

{

title: '文件预览',

formatter: function(value, row, index) {

return '<a href="javascript:downloadFile(' + row.fileId + ')"><img style="width:30;height:30px;" src="/profile/upload/' + row.filePath + '"/></a>';

}

},

注意：如果只是单纯的上传一张图片没有其他参数可以使用通用方法 /common/upload  
请求处理方法 com.numberone.web.controller.common.CommonController

**下载实现流程**

1、参考示例代码。

function downloadFile(fileId){

window.location.href = ctx + "system/sysFile/downloadFile/" + fileId;

}

2、在Controller添加对应上传方法

@GetMapping("/downloadFile/{fileId}")

public void downloadFile(@PathVariable("fileId") Integer fileId, HttpServletResponse response) throws Exception

{

SysFile sysFile = sysFileService.selectSysFileById(fileId);

String filePath = sysFile.getFilePath();

String realFileName = sysFile.getFileName() + filePath.substring(filePath.indexOf("."));

String path = Global.getUploadPath() + sysFile.getFilePath();

response.setCharacterEncoding("utf-8");

response.setContentType("multipart/form-data");

response.setHeader("Content-Disposition", "attachment;fileName=" + realFileName);

FileUtils.writeBytes(path, response.getOutputStream());

}

[**事务管理**](http://doc.ruoyi.vip/#/standard/htsc?id=%e4%ba%8b%e5%8a%a1%e7%ae%a1%e7%90%86)

在Spring Boot中，当我们使用了spring-boot-starter-jdbc或spring-boot-starter-data-jpa依赖的时候，框架会自动默认分别注入DataSourceTransactionManager或JpaTransactionManager。 所以我们不需要任何额外配置就可以用@Transactional注解进行事务的使用。

例如：新增用户时需要插入用户表、用户与岗位关联表、用户与角色关联表。就可以使用事务让它实现回退。  
做法非常简单，我们只需要在方法上添加@Transactional注解即可。事务可以用于Service和Controller

@Transactional

public int insertUser(User user)

{

// 新增用户信息

int rows = userMapper.insertUser(user);

// 新增用户岗位关联

insertUserPost(user);

// 新增用户与角色管理

insertUserRole(user);

return rows;

}

常见坑点1：遇到非检测异常时，事务开启，也无法回滚。 例如下面这段代码，账户余额依旧增加成功，并没有因为后面遇到检测异常而回滚！！

@Transactional

public void addMoney() throws Exception {

//先增加余额

accountMapper.addMoney();

//然后遇到故障

throw new SQLException("发生异常了..");

}

原因分析：因为Spring的默认的事务规则是遇到运行异常（RuntimeException）和程序错误（Error）才会回滚。如果想针对非检测异常进行事务回滚，可以在@Transactional 注解里使用 rollbackFor 属性明确指定异常。例如下面这样，就可以正常回滚：

@Transactional(rollbackFor = Exception.class)

public void addMoney() throws Exception {

//先增加余额

accountMapper.addMoney();

//然后遇到故障

throw new SQLException("发生异常了..");

}

常见坑点2： 在业务层捕捉异常后，发现事务不生效。 这是许多新手都会犯的一个错误，在业务层手工捕捉并处理了异常，你都把异常“吃”掉了，Spring自然不知道这里有错，更不会主动去回滚数据。 例如：下面这段代码直接导致增加余额的事务回滚没有生效。

@Transactional

public void addMoney() throws Exception {

//先增加余额

accountMapper.addMoney();

//谨慎：尽量不要在业务层捕捉异常并处理

try {

throw new SQLException("发生异常了..");

} catch (Exception e) {

e.printStackTrace();

}

}

推荐做法：在业务层统一抛出异常，然后在控制层统一处理。

@Transactional

public void addMoney() throws Exception {

//先增加余额

accountMapper.addMoney();

//推荐：在业务层将异常抛出

throw new RuntimeException("发生异常了..");

}

Transactional注解的常用属性表：

| **属性** | **说明** |
| --- | --- |
| propagation | 事务的传播行为，默认值为 REQUIRED。 |
| isolation | 事务的隔离度，默认值采用 DEFAULT |
| timeout | 事务的超时时间，默认值为-1，不超时。如果设置了超时时间(单位秒)，那么如果超过该时间限制了但事务还没有完成，则自动回滚事务。 |
| read-only | 指定事务是否为只读事务，默认值为 false；为了忽略那些不需要事务的方法，比如读取数据，可以设置 read-only 为 true。 |
| rollbackFor | 用于指定能够触发事务回滚的异常类型，如果有多个异常类型需要指定，各类型之间可以通过逗号分隔。{xxx1.class, xxx2.class,……} |
| noRollbackFor | 抛出 no-rollback-for 指定的异常类型，不回滚事务。{xxx1.class, xxx2.class,……} |
| .... |  |

TransactionDefinition传播行为的常量：

| **常量** | **含义** |
| --- | --- |
| TransactionDefinition.PROPAGATION\_REQUIRED | 如果当前存在事务，则加入该事务；如果当前没有事务，则创建一个新的事务。这是默认值。 |
| TransactionDefinition.PROPAGATION\_REQUIRES\_NEW | 创建一个新的事务，如果当前存在事务，则把当前事务挂起。 |
| TransactionDefinition.PROPAGATION\_SUPPORTS | 如果当前存在事务，则加入该事务；如果当前没有事务，则以非事务的方式继续运行。 |
| TransactionDefinition.PROPAGATION\_NOT\_SUPPORTED | 以非事务方式运行，如果当前存在事务，则把当前事务挂起。 |
| TransactionDefinition.PROPAGATION\_NEVER | 以非事务方式运行，如果当前存在事务，则抛出异常。 |
| TransactionDefinition.PROPAGATION\_MANDATORY | 如果当前存在事务，则加入该事务；如果当前没有事务，则抛出异常。 |
| TransactionDefinition.PROPAGATION\_NESTED | 如果当前存在事务，则创建一个事务作为当前事务的嵌套事务来运行；如果当前没有事务，则该取值等价于TransactionDefinition.PROPAGATION\_REQUIRED。 |

提示：事务的传播机制是指如果在开始当前事务之前，一个事务上下文已经存在，此时有若干选项可以指定一个事务性方法的执行行为。 即:在执行一个@Transactinal注解标注的方法时，开启了事务；当该方法还在执行中时，另一个人也触发了该方法；那么此时怎么算事务呢，这时就可以通过事务的传播机制来指定处理方式。

[**异常处理**](http://doc.ruoyi.vip/#/standard/htsc?id=%e5%bc%82%e5%b8%b8%e5%a4%84%e7%90%86)

在日常开发中程序发生了异常，往往需要通过一个统一的异常处理，来保证客户端能够收到友好的提示。  
通常情况下我们用try..catch..对异常进行捕捉处理，但是在实际项目中对业务模块进行异常捕捉，会造成代码重复和繁杂， 我们希望代码中只有业务相关的操作，所有的异常我们单独设立一个类来处理它。全局异常就是对框架所有异常进行统一管理 而这就表示在框架需要一个机制，将程序的异常转换为用户可读的异常。而且最重要的，是要将这个机制统一，提供统一的异常处理。 我们在可能发生异常的方法，全部throw抛给前端控制器；然后由前端控制器调用 全局异常处理器 对异常进行统一处理。 如此，我们现在的Controller中的方法就可以很简洁了。

1、统一返回实体定义

package com.numberone.common.core.domain;

import java.util.HashMap;

/\*\*

\* 操作消息提醒

\*

\* @author numberone

\*/

public class AjaxResult extends HashMap<String, Object>

{

private static final long serialVersionUID = 1L;

/\*\*

\* 返回错误消息

\*

\* @param code 错误码

\* @param msg 内容

\* @return 错误消息

\*/

public static AjaxResult error(String msg)

{

AjaxResult json = new AjaxResult();

json.put("msg", msg);

json.put("code", 500);

return json;

}

/\*\*

\* 返回成功消息

\*

\* @param msg 内容

\* @return 成功消息

\*/

public static AjaxResult success(String msg)

{

AjaxResult json = new AjaxResult();

json.put("msg", msg);

json.put("code", 0);

return json;

}

}

2、定义登录异常定义

package com.numberone.common.exception;

/\*\*

\* 登录异常

\*

\* @author numberone

\*/

public class LoginException extends RuntimeException

{

private static final long serialVersionUID = 1L;

protected final String message;

public LoginException(String message)

{

this.message = message;

}

@Override

public String getMessage()

{

return message;

}

}

3、基于@ControllerAdvice注解的Controller层的全局异常统一处理

package com.numberone.framework.web.exception;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

import org.springframework.web.bind.annotation.RestControllerAdvice;

import com.numberone.common.core.domain.AjaxResult;

import com.numberone.common.exception.LoginException;

/\*\*

\* 全局异常处理器

\*

\* @author numberone

\*/

@RestControllerAdvice

public class GlobalExceptionHandler

{

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

/\*\*

\* 登录异常

\*/

@ExceptionHandler(LoginException.class)

public AjaxResult loginException(LoginException e)

{

log.error(e.getMessage(), e);

return AjaxResult.error(e.getMessage());

}

}

4、测试访问请求

@Controller

public class SysIndexController

{

/\*\*

\* 首页方法

\*/

@GetMapping("/index")

public String index(ModelMap mmap)

{

/\*\*

\* 模拟用户未登录，抛出业务逻辑异常

\*/

SysUser user = ShiroUtils.getSysUser();

if (StringUtils.isNull(user))

{

throw new LoginException("用户未登录，无法访问请求。");

}

mmap.put("user", user);

return "index";

}

}

根据上面代码含义，当我们在访问/index时就会发生LoginException业务逻辑异常，按照我们之前的全局异常配置以及统一返回实体实例化，访问后会出现AjaxResult格式JSON数据， 下面我们运行项目访问查看效果。  
界面输出内容如下所示：

{

"msg": "用户未登录，无法访问请求。",

"code": 500

}

这个代码示例写的非常浅显易懂，但是需要注意的是：基于@ControllerAdvice注解的全局异常统一处理只能针对于Controller层的异常，意思是只能捕获到Controller层的异常， 在service层或者其他层面的异常都不能捕获。

若依系统的全局异常处理器GlobalExceptionHandler  
注意：如果全部异常处理返回json，那么可以使用@RestControllerAdvice代替@ControllerAdvice，这样在方法上就可以不需要添加@ResponseBody。

[**系统日志**](http://doc.ruoyi.vip/#/standard/htsc?id=%e7%b3%bb%e7%bb%9f%e6%97%a5%e5%bf%97)

在实际开发中，对于某些关键业务，我们通常需要记录该操作的内容，一个操作调一次记录方法，每次还得去收集参数等等，会造成大量代码重复。 我们希望代码中只有业务相关的操作，所有的异常我们单独设立一个注解来处理它。

在需要被记录日志的controller方法上添加@Log注解，使用方法如下：

@Log(title = "用户管理", businessType = BusinessType.INSERT)

支持参数如下：

| **参数** | **类型** | **默认值** | **描述** |
| --- | --- | --- | --- |
| title | String | 空 | 操作模块 |
| businessType | BusinessType | BusinessType.OTHER | 操作功能 |
| operatorType | OperatorType | OperatorType.MANAGE | 操作人类别 |
| isSaveRequestData | boolean | true | 是否保存请求的参数 |

逻辑实现代码 com.numberone.framework.aspectj.LogAspect  
查询操作详细记录可以登录系统（系统管理-操作日志）

[**数据权限**](http://doc.ruoyi.vip/#/standard/htsc?id=%e6%95%b0%e6%8d%ae%e6%9d%83%e9%99%90)

在实际开发中，需要设置用户只能查看哪些部门的数据，一般称为数据权限  
默认系统管理员admin拥有所有数据权限（userId=1）  
在需要数据权限控制方法上添加@DataScope注解  
@DataScope(tableAlias = "u")，其中u用来表示表的别名

/\*\* 表的别名 \*/

String tableAlias() default "";

在mybatis查询标签中添加数据范围过滤 ${params.dataScope}

会生成如下关键代码：

select u.user\_id, u.dept\_id, u.login\_name, u.user\_name, u.email , u.phonenumber,

u.password, u.sex, u.avatar, u.salt, u.status, u.del\_flag, u.login\_ip,

u.login\_date, u.create\_by, u.create\_time, u.remark, d.dept\_name

from sys\_user u

left join sys\_dept d on u.dept\_id = d.dept\_id

where u.del\_flag = '0'

and u.dept\_id in ( select dept\_id from sys\_role\_dept where role\_id = 2 )

逻辑实现代码 com.numberone.framework.aspectj.DataScopeAspect

[**多数据源**](http://doc.ruoyi.vip/#/standard/htsc?id=%e5%a4%9a%e6%95%b0%e6%8d%ae%e6%ba%90)

在实际开发中，经常可能遇到在一个应用中可能需要访问多个数据库的情况  
在需要切换数据源Service或Mapper方法上添加@DataSource注解  
@DataSource(value = DataSourceType.MASTER)，其中value用来表示数据源名称

/\*\* 切换数据源名称 \*/

public DataSourceType value() default DataSourceType.MASTER;

注解实现数据源切换

@DataSource(value = DataSourceType.MASTER)

public List<SysUser> selectUserList(SysUser user)

{

return userMapper.selectUserList(user);

}

手动实现数据源切换

public List<SysUser> selectUserList(SysUser user)

{

DynamicDataSourceContextHolder.setDataSourceType(DataSourceType.MASTER.name());

List<SysUser> userList = userMapper.selectUserList(user);

DynamicDataSourceContextHolder.clearDataSourceType();

return userList;

}

逻辑实现代码 com.numberone.framework.aspectj.DataSourceAspect

注意：目前配置了一个从库，默认关闭状态。可新增多个从库，支持不同数据源（Mysql、Oracle、SQLServer）

[**代码生成**](http://doc.ruoyi.vip/#/standard/htsc?id=%e4%bb%a3%e7%a0%81%e7%94%9f%e6%88%90)

大部分项目里其实有很多代码都是重复的，几乎每个基础模块的代码都有增删改查的功能，而这些功能都是大同小异，如果这些功能都要自己去写，将会大大浪费我们的精力降低效率。所以这种重复性的代码可以使用代码生成。 1、修改代码生成配置  
编辑resources目录下的generator.yml  
author: # 开发者姓名，生成到类注释上  
packageName: # 默认生成包路径  
autoRemovePre: # 是否自动去除表前缀  
tablePrefix: # 表前缀

2、新建数据库表结构（需要表注释）

drop table if exists sys\_test;

create table sys\_test (

test\_id int(11) auto\_increment comment '测试id',

test\_name varchar(30) default '' comment '测试名称',

primary key (test\_id)

) engine=innodb auto\_increment=1 default charset=utf8 comment = '测试表';

3、登录系统-系统工具 -> 代码生成  
找到sys\_test表，点击生成代码会得到一个numberone.zip  
执行sql文件，覆盖文件到对应目录即可

所有代码生成的相关业务逻辑代码在numberone-generator模块，可以自行调整或剔除

[**定时任务**](http://doc.ruoyi.vip/#/standard/htsc?id=%e5%ae%9a%e6%97%b6%e4%bb%bb%e5%8a%a1)

在实际项目开发中，除了Web应用、还有一类不可缺少的，那就是定时任务。 定时任务的场景可以说非常广泛，比如某些视频网站，购买会员后，每天会给会员送成长值，每月会给会员送一些电影券； 比如在保证最终一致性的场景中，往往利用定时任务调度进行一些比对工作；比如一些定时需要生成的报表、邮件；比如一些需要定时清理数据的任务等。 所有我们提供方便友好的web界面，实现动态管理任务，可以达到动态控制定时任务启动、暂停、重启、删除、添加、修改等操作，极大地方便了开发过程。

1、新建定时任务信息（系统监控 -> 定时任务）  
任务名称：对应后台bean注解名称，如@Component("ryTask")  
任务组名：对应的定时任务组名 随意填写。  
方法名称：对应后台任务方法名称如ryParams  
方法参数：对应后台任务方法名称值如ry，没有可不填。  
执行表达式：可查询官方cron表达式介绍  
执行策略：  
立即执行（所有被misfire的执行会被立即执行，然后按照正常调度继续执行trigger。 9点和10点的被忽略掉，好像什么都没发生一样。下次执行将在11点被执行。） 执行一次（立即执行第一次misfire的操作，并且放弃其他misfire的（类似所有misfire的操作被合并执行了）。然后继续按调度执行。无论misfire多少次trigger的执行，都只会立刻执行1次 9点和10点的被合并执行一次（换句话说，10点需要执行的那次，被pass了）。下次执行将在11点被准时执行）  
放弃执行（所有被misfire的执行都被忽略掉，调度器会像平时一样等待下次调度 9点和10点的执行（misfire的2个）被立即执行，下次执行将在11点被准时执行。）  
并发执行：是否需要多个任务间同时执行  
状态：是否启动定时任务  
备注：描述信息

2、点击执行一次，测试定时任务是否正常及调度日志是否正确记录

所有定时任务的相关业务逻辑代码在numberone-quartz模块，可以自行调整或剔除