<LCP框架>

开发指导手册

## 文档控制

**更改记录**

4

| 日期 | 作者 | 版本 | 更改参考 |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

内容目录

[文档控制 2](#_Toc472587999)

[1. LCP介绍 4](#_Toc472588000)

[2. 环境搭建及项目配置 5](#_Toc472588001)

[2.1. 开发工具 5](#_Toc472588002)

[2.2. 工程创建 5](#_Toc472588003)

[2.3. 编译项目 5](#_Toc472588004)

[2.4. 创建数据库,并初始化 5](#_Toc472588005)

[2.5. 配置JNDI数据源 6](#_Toc472588006)

[2.6. 安装并启动redis 6](#_Toc472588007)

[2.7. 运行项目 6](#_Toc472588008)

[3. LCP开源组件介绍 7](#_Toc472588009)

[3.1. Spring Security 7](#_Toc472588010)

[3.2. Quartz 7](#_Toc472588011)

[3.3. Activiti 7](#_Toc472588012)

[3.4. CXF 7](#_Toc472588013)

[3.5. Redis 8](#_Toc472588014)

[3.6. LiquiBase 8](#_Toc472588015)

[4. LCP常用功能 9](#_Toc472588016)

[4.1. Lov配置及使用 9](#_Toc472588017)

[4.2. 快速编码的配置和使用 9](#_Toc472588018)

[4.3. 描述维护的配置和使用 10](#_Toc472588019)

[4.4. 代码生成器使用 10](#_Toc472588020)

[4.5. 代码生成器使用 14](#_Toc472588021)

[5. 后端开发 18](#_Toc472588022)

[5.1. Controller层 18](#_Toc472588023)

[5.2. Service层 19](#_Toc472588024)

[5.3. DTO层 20](#_Toc472588025)

[6. 前端开发 21](#_Toc472588026)

[6.1. 常用组件使用 21](#_Toc472588027)

[6.2. 常用函数介绍 30](#_Toc472588028)

[7. 入门案例 32](#_Toc472588029)

[7.1. 实例介绍 32](#_Toc472588030)

[7.2. 后台开发 32](#_Toc472588031)

[7.3. 前台开发 37](#_Toc472588032)

[8. 前端开发案例 54](#_Toc472588033)

[8.1. 单表操作 54](#_Toc472588034)

[8.2. 头行结构 66](#_Toc472588035)

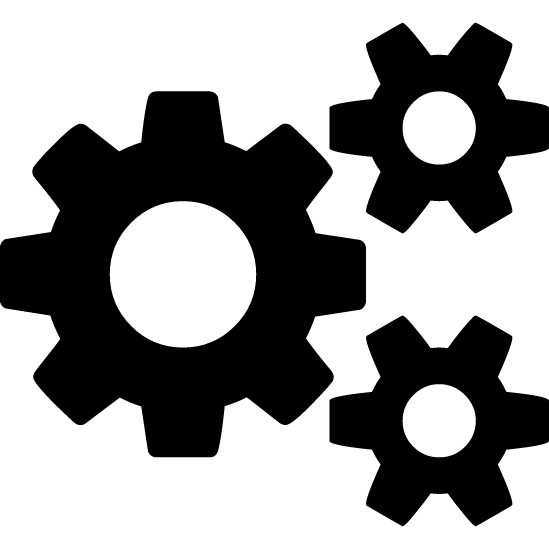
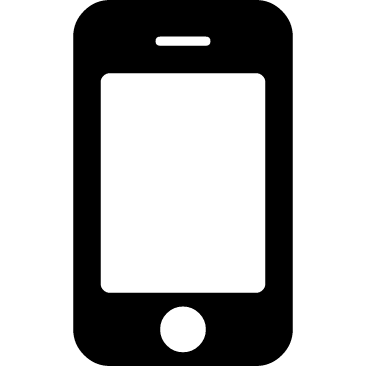
[9. 常见问题及解决办法 67](#_Toc472588036)

[未结与已结问 68](#_Toc472588037)

[未结问题 68](#_Toc472588038)

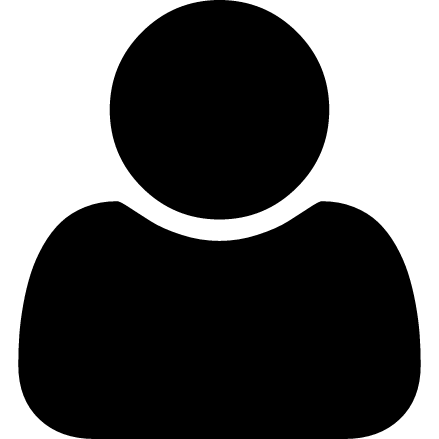
[已结问题 68](#_Toc472588039)

## LCP介绍

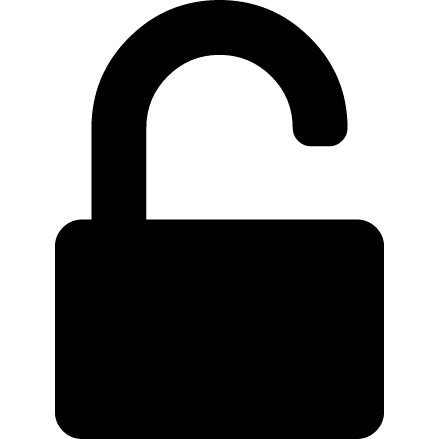


管理  
WEBUI

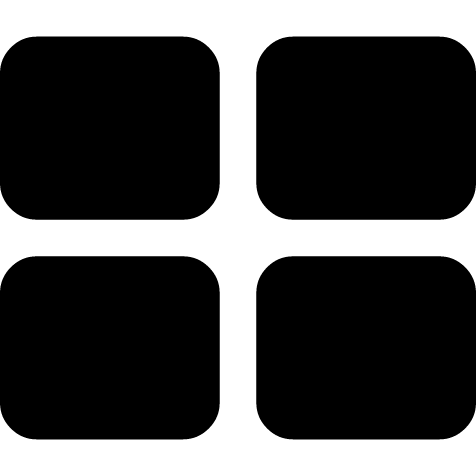
用户



权限



基础



**REST**

**API**

**REST**

**API**

**REST**

**API**

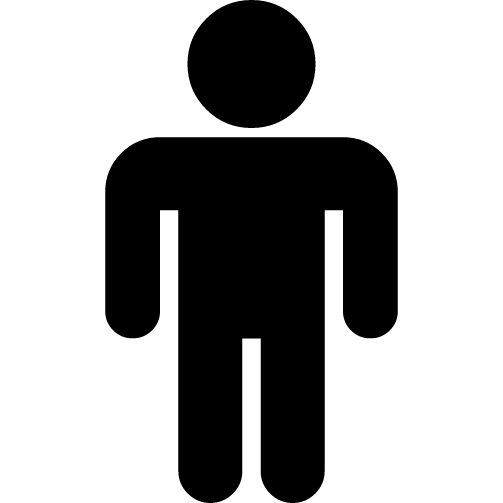
**REST**

**API**

**REST**

**API**

HAP



* + 基于Java流行开源技术构建 LCP使用业界最流行的开源技术搭建，框架本身也完全开源，做到真正的技术开放。
  + 前后端分离。具有清晰的模块界限、服务独立部署和技术多样性的能力特点。
  + 满足传统Web和分布式应用开发 框架以适应经典的MVC架构和分布式大型应用开发为设计目标，两种风格的开发和部署无缝切换。
  + 满足多平台移动应用开发 使用H5和JS混合技术，配合移动中台管理组件，构建企业级跨平台移动应用解决方案。

## 环境搭建及项目配置

### 开发工具

JDK 1.8 及以上

IntelliJ IDEA 2017及以上

redis 3.0 及以上

maven 3.3 及以上

Tomcat 7+ (不支持 Tomcat 6)

数据库（任选）

* Mysql server 5.6 及以上
* SqlServer 2012 及以上
* Oracle 9 以上

Navicat Premium（或其他）

Svn 1.8+

### 工程创建

参考[《东风物流知识库》](https://rdc.hand-china.com/gitlab/HAP/hap-developer-guide/blob/master/chapter1.2.md)“快速入门”章节。

### 编译项目

参考《东风物流知识库》“快速入门”章节。

### 创建数据库,并初始化

参考《东风物流知识库》“快速入门”章节。

### 配置JNDI数据源

参考《东风物流知识库》“快速入门”章节。

### 安装并启动redis

参考《东风物流知识库》“快速入门”章节。

### 运行项目

参考《东风物流知识库》“快速入门”章节。

## LCP开源组件介绍

LCP开源组件与LCP版本无关，下列主要开源组件使用在LCP每个版本均适用。

### Spring Security

LCP中的用户权限控制,用户认证以及系统安全,通过Spring Security实现.

### Quartz

Quartz是一个强大的开源调度框架,LCP中的任务调度模块由Quartz实现,其中通过Quartz可以为系统增加定时任务,其中通过cronExpression定义的定时任务,相当灵活,也是开发过程中经常使用的.

### Activiti

LCP中集成了开源的工作流框架Activiti.用于实现工作流.

### CXF

Web Service是分布式系统实现消息传递和远程调用的技术,可以通过Web Service技术为外部系统提供可靠的外部接口,Apache CXF框架支持多种协议的Web Service实现,并且支持多种传输协议.LCP集成了CXF.实现WebService的发布.

### Redis

Redis是一个开源的使用C语言编写的内存非关系型数据库,采用单线程IO复用模型实现.LCP使用Redis进行数据缓存,Session管理以及消息订阅发布的实现.Redis是典型的CS结构.存在Client和Server两个部分,Server负责管理内存数据,Client为管理数据提供接口.LCP使用Jedis作为客户端与Redis Server交互,通过继承Spring Data Redis简化开发难度.

### LiquiBase

LiquiBase是一个日志式的数据库变更管理工具.通过记录下每次数据库的变更操作,可以方便的坐到数据库的持续集成.

## LCP常用功能

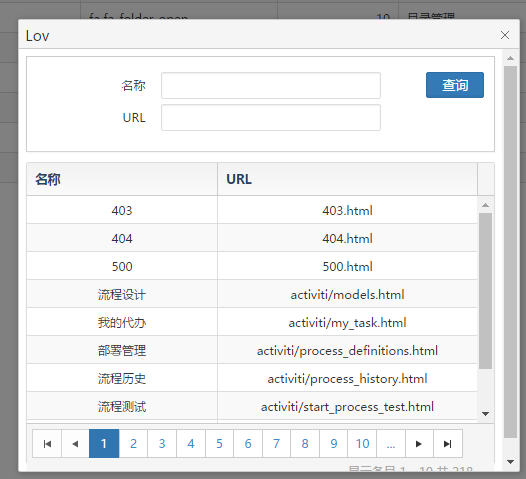
下列常用功能适用于LCP2.0及以上

### Lov配置及使用

在一些场景下需要从特定的范围内取得某条记录的主键,并且支持有条件的查询



点击之后出现弹出框



然后选中其中的某条记录,

### 快速编码的配置和使用

代码维护用于维护开发过程中遇到的编码信息.在系统管理->代码维护中,



快速编码用于维护类似java枚举值的描述信息.

<script src="${base.contextPath}/common/code?resourceTypeData=SYS.RESOURCE\_TYPE " type="text/javascript"></script>

通过script标签引用.

可以获得如下全局变量

var resourceTypeData=[{"objectVersionNumber":1,"codeId":10008,"codeValueId":10001,"description":"HTML页面","meaning":"HTML页面","value":"HTML","orderSeq":10},{"objectVersionNumber":1,"codeId":10008,"codeValueId":10002,"description":"URL服务","meaning":"URL服务","value":"SERVICE","orderSeq":10}];

然后通过全局变量引用

### 描述维护的配置和使用

界面中的描述信息可以通过描述维护来提供多语言显示,在系统配置->描述维护中,



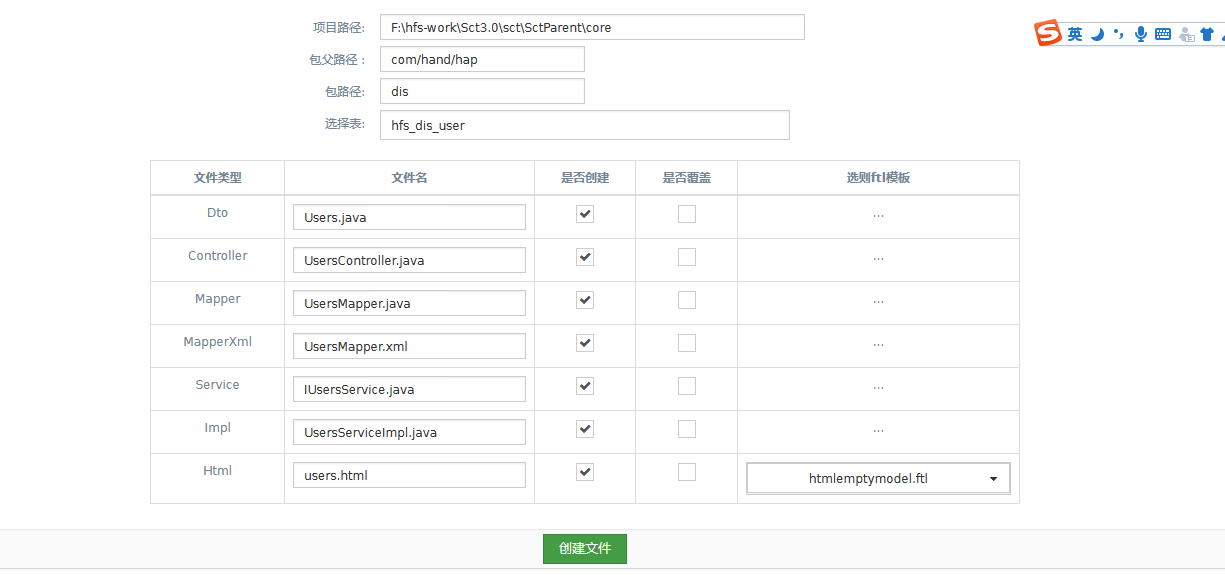
按图中的方式新建描述维护信息,在html中通过freemarker标签使用,如

<@spring.message "hap.new"/>,红色部分为描述维护的代码.

### 代码生成器使用

代码生成器url为：<http://XX/generator/generator.html>

打开后如下图：



我们根据要求输入地址及名称：

项目路径：输入当前项目路径F:\hfs-work\Sct3.0\sct\SctParent\core

包父路径：路径为：com/hand/hap 注意不可写成com\hand\hap，否则生成文件会出错。

包父类：dis

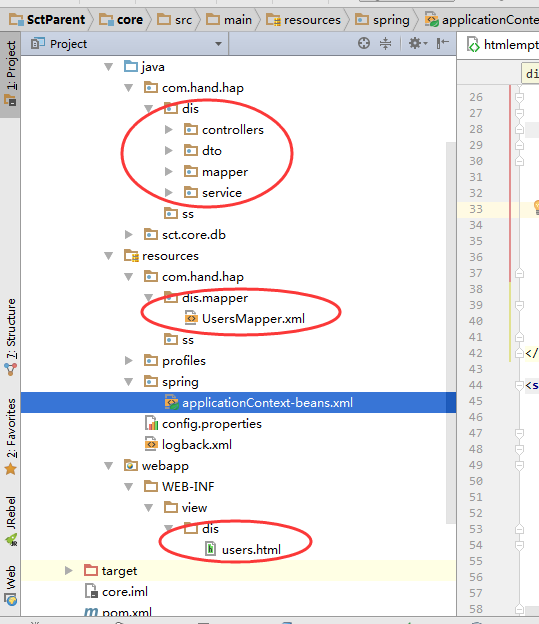
表名称：hfs\_dis\_sct

默认的文件名称为表名后两个单词，如disUser.html，一般按照要规范是取最后一个单词作为类名称

Html模板页面选择是写死，后续如果有其他模板加入，可以将其维护起来

一切按要求填写完后点击创建文件

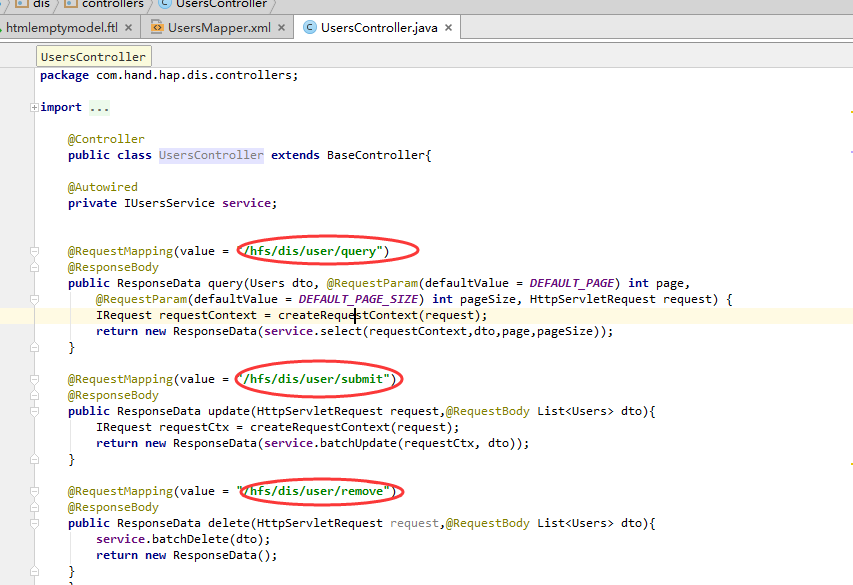
#### 查看生成文件



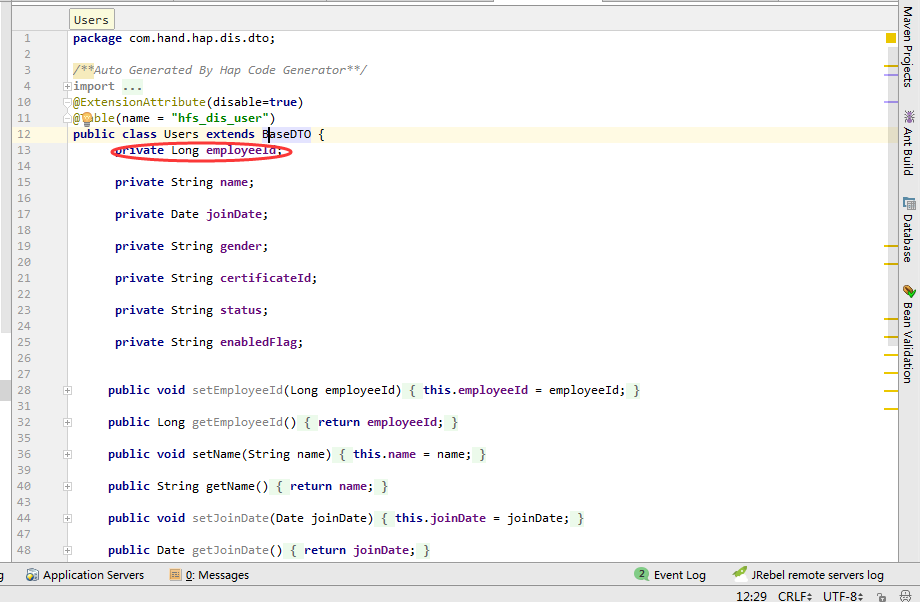
看出各类文件已经生成。

Controller文件：

url为表名称加上submit或者delete，url一般从模块名称值，如下图的hfs是不需要的

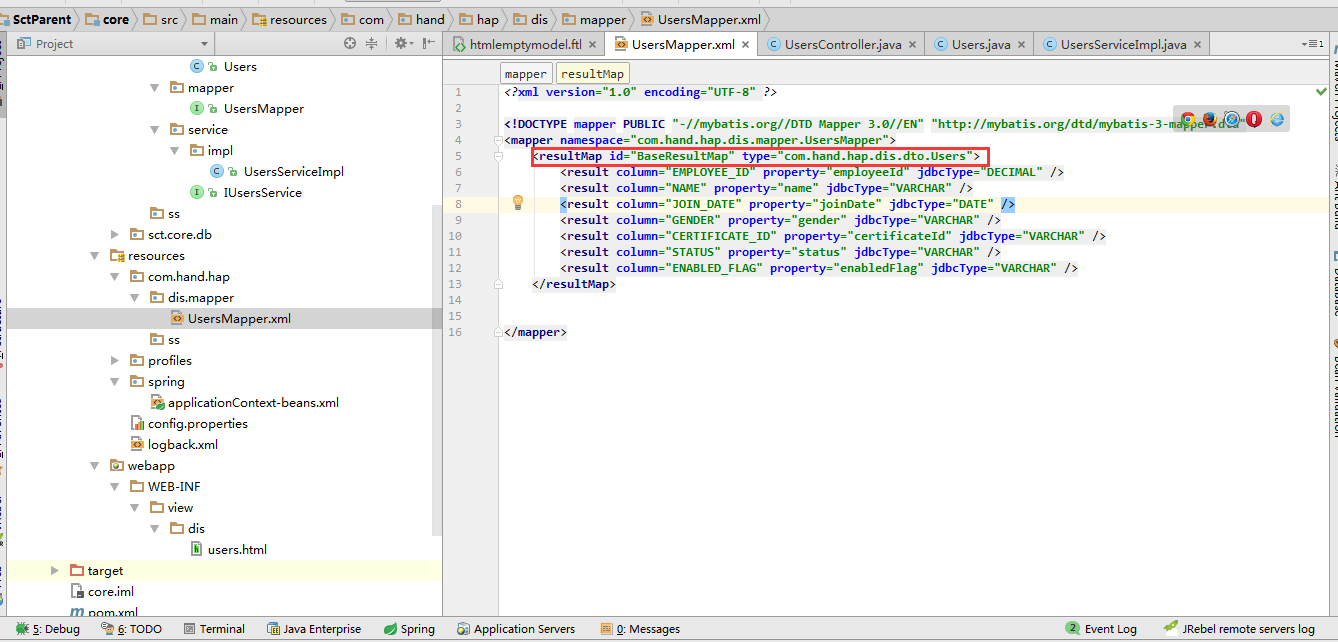


#### dto



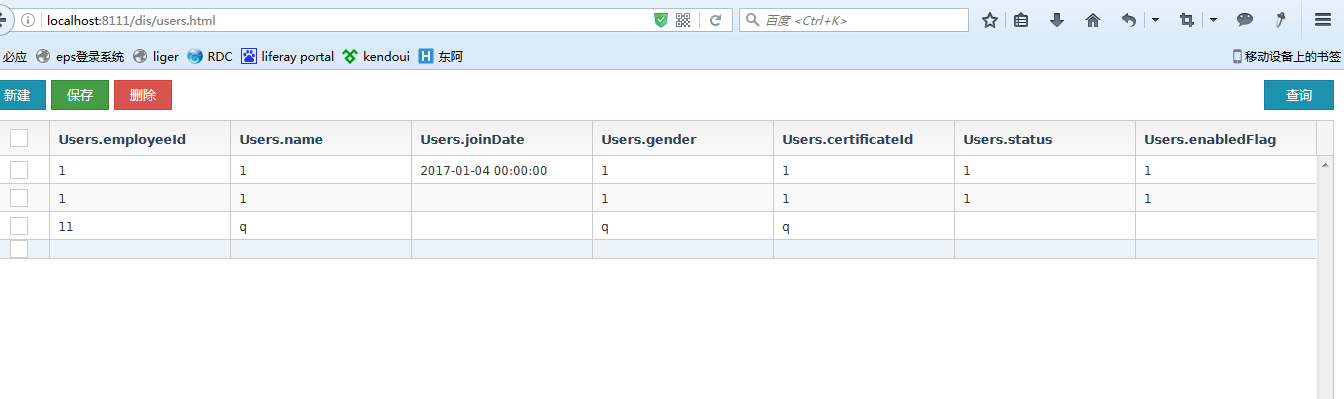
已经继承basedto，但是主键等特殊需求注解需要自己添加。

#### mapper.xml



需要自己手动将stdmapper.xml导入，后续有待完善

#### html



生成的HTML样式如图，需要特定进行修改，到此，就实现了页面简单的增删改查

### Excel通用导出使用

#### 前端-按钮的设计和实现

先在前端HTML代码上定义一个【导出Excel】按钮

<div class="pull-left" id="toolbar-btn" style="padding-bottom:10px;">

<span class="btn btn-primary" style="float:left;margin-right:5px;"

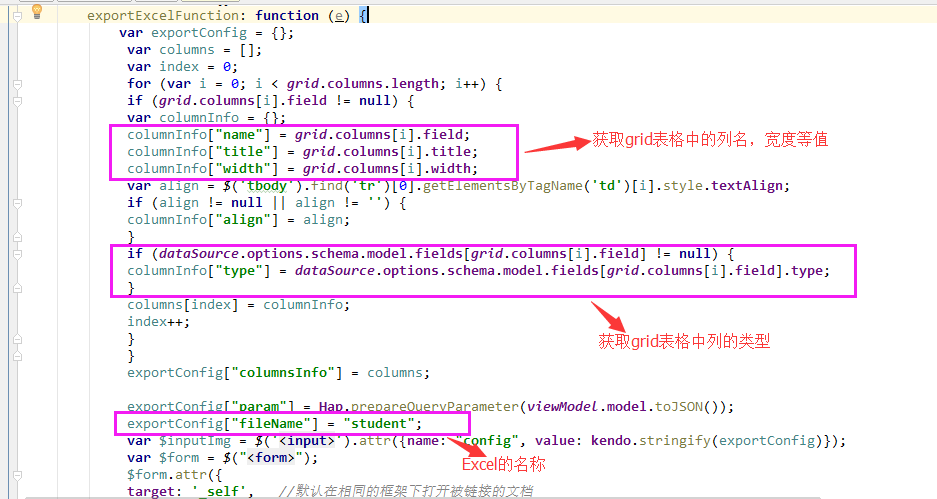
data-bind="click:exportExcelStudent"><i class="fa fa-file-excel-o" style="margin-right:3px;"></i><@spring.message "hap.exportexcel"/></span>

</div>

<!--将按钮和viewmodel进行绑定-->

<script>kendo.bind($('#toolbar-btn'), viewModel);</script>

#### 前端-实现按钮的exportExcelStudent方法



var viewModel = kendo.observable({

model : {},

exportExcelFunction: function (e) {

var exportConfig = {};

var columns = [];

var index = 0;

for (var i = 0; i < grid.columns.length; i++) {

if (grid.columns[i].field != null) {

var columnInfo = {};

columnInfo["name"] = grid.columns[i].field;

columnInfo["title"] = grid.columns[i].title;

columnInfo["width"] = grid.columns[i].width;

var align = $('tbody').find('tr')[0].getElementsByTagName('td')[i].style.textAlign;

if (align != null || align != '') {

columnInfo["align"] = align;

}

if (dataSource.options.schema.model.fields[grid.columns[i].field] != null) {

columnInfo["type"] = dataSource.options.schema.model.fields[grid.columns[i].field].type;

}

columns[index] = columnInfo;

index++;

}

}

exportConfig["columnsInfo"] = columns;

exportConfig["param"] = Hap.prepareQueryParameter(viewModel.model.toJSON());

exportConfig["fileName"] = "student";

var $inputImg = $('<input>').attr({name: "config", value: kendo.stringify(exportConfig)});

var $form = $("<form>");

$form.attr({

target: '\_self', //默认在相同的框架下打开被链接的文档

method: 'post',

action: '${base.contextPath}/sys/student/export' //绑定后台方法

}).append($inputImg);

$form.submit();

}

})

#### 后台-实现EXCEL导出方法



@RequestMapping(value = "/sys/student/export")

public void createXLSForStudent(HttpServletRequest request, @RequestParam String config,

HttpServletResponse httpServletResponse) {

IRequest requestContext = createRequestContext(request);

try {

JavaType type = objectMapper.getTypeFactory().constructParametrizedType(ExportConfig.class,

ExportConfig.class, Student.class, ColumnInfo.class);

ExportConfig<Function, ColumnInfo> exportConfig = objectMapper.readValue(config, type);

excelService.exportAndDownloadExcel("com.hand.hap.system.mapper.StudentMapper.findStudent",

exportConfig, request, httpServletResponse, requestContext);

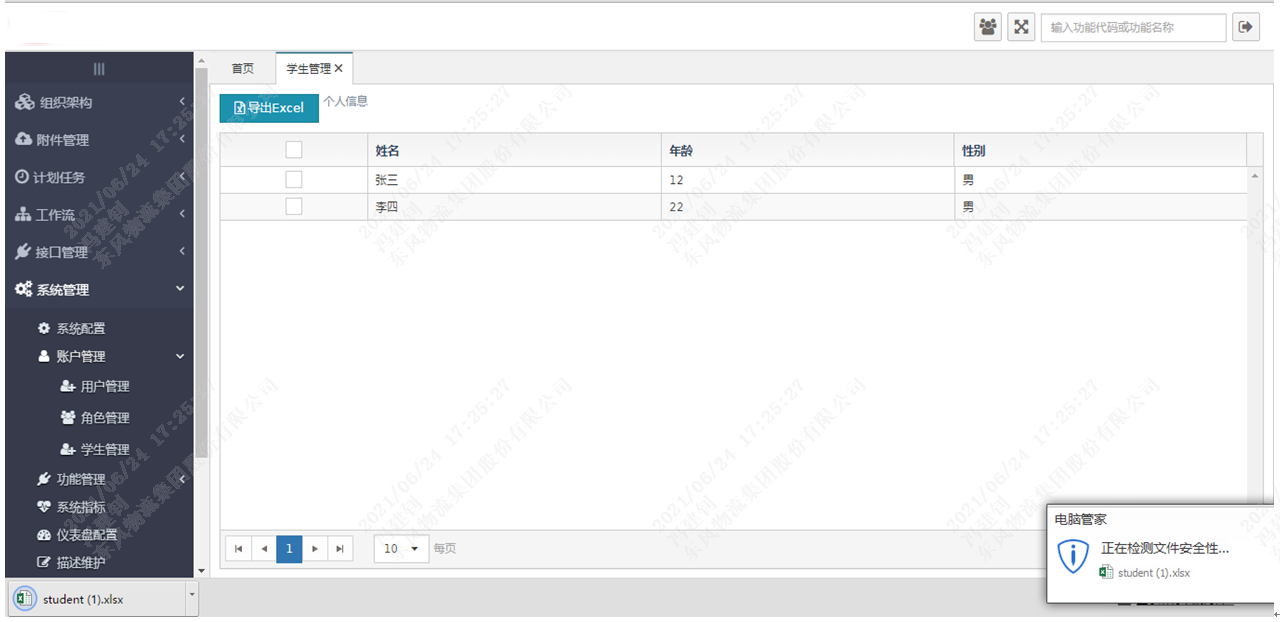
} catch (IOException e) {

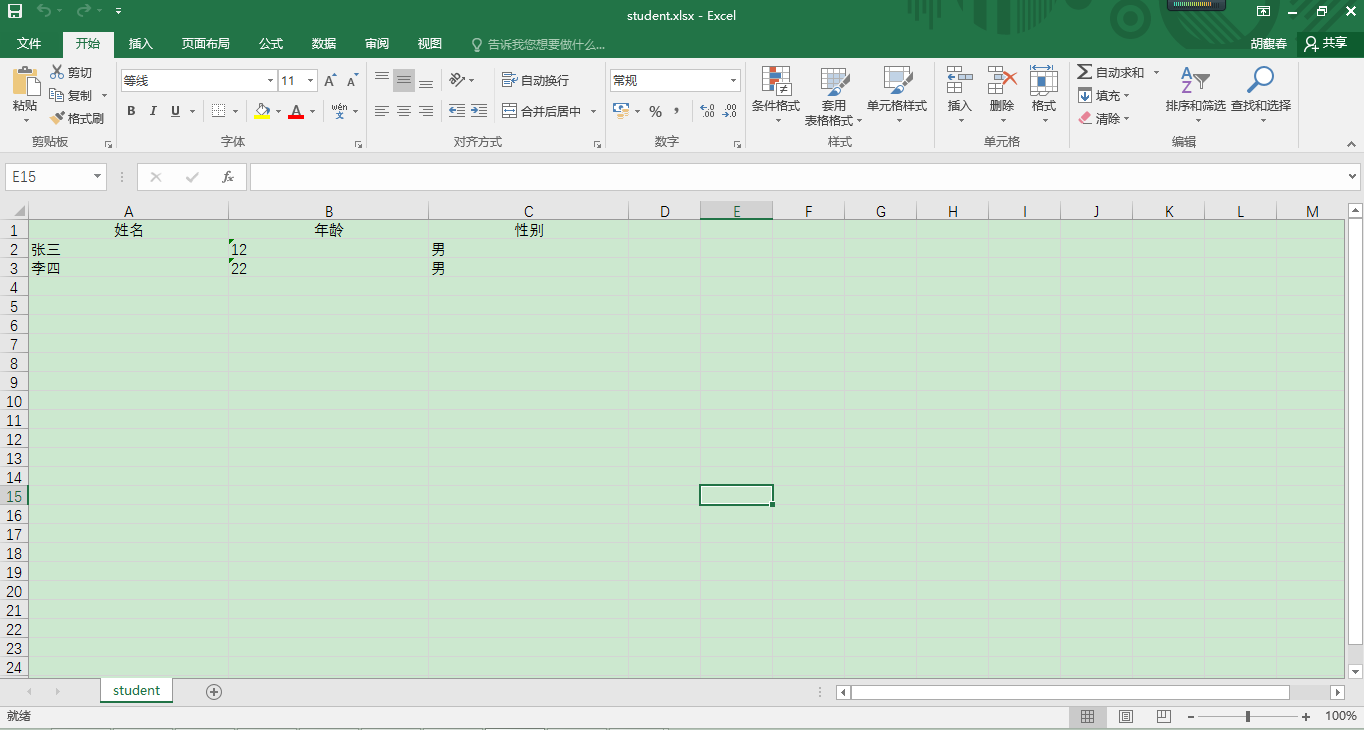
e.printStackTrace();

}

}

#### 显示效果展示





## 后端开发

### Controller层

#### BaseController简介

BaseController是一个由LCP框架封装好的Controller类。其中定义了一些常用的方法。如获取用户Id，获取用户角色，当前语言环境，异常捕获处理等。

#### BaseController常用属性介绍

* DEFAULT\_PAGE：设置分页默认页码，默认值为1。
* DEFAULT\_PAGE\_SIZE：设置默认分页每页显示的记录条数，默认值为10。
* MessageSource messageSource：消息源。其中getMessage(String code, Object[] args, Locale locale)可以取得系统中当前语言环境下在描述维护中维护的code对应的提示消息。（描述维护是LCP框架的一个功能，用于维护同一代码在不同语言环境下的不同显示状态。）

#### BaseController常用方法介绍

* checkToken():用于发布webservice时做token校验。BaseController中重载了该方 法。分别是：checkToken(HttpServletRequest request, Collection<? extends BaseDTO> dtos)

checkToken(HttpSession session, Collection<? extends BaseDTO> dtos)

checkToken(HttpServletRequest request, BaseDTO baseDTO)

checkToken(HttpSession session, BaseDTO baseDTO)

实际开发中结合自己的需要调用即可，该方法返回值为空。

* exceptionHandler(Exception exception, HttpServletRequest request)：用于处理控制层所有的异常。当然你也可以在自己的Controller层捕获处理自己的异常。如果继承了BaseController类，自己的控制层未做处理，而是选择将异常抛出，那么异常就会在此处被拦截。
* getErrorMessage(Errors errors, HttpServletRequest request)：用于获取异常消息。返回的是根据当前语言环境翻译后的错误消息，errors是包含错误的对象。
* nls(HttpServletRequest request, String code, Object[] args)：也是用于获取描述维护中当前语言环境下的消息。通常用于自定义的是错误消息。code是描述维护中消息的代码。底层实现用的是messageSource.getMessage(code, args, code, locale); args是message中需要用到的参数。如果没有直接传null即可。也可以直接调用nls(HttpServletRequest request, String code)。该方法中没有消息参数。
* getStandardFieldErrorMessage(FieldErrorWithBean fieldError, Locale locale)：该方法用于获取DTO字段校验的标准错误消息，如非空验证等。FieldErrorWithBean可直接附加field 所属的 bean，fieldError可直接获取目标bean。local为当前语言环境。
* findDeclareClass(Class fromClass, String fieldName)：用于找到当前属性来之那个父类。fromClass为当前类，fieldName为当前属性名。
* getUserId(HttpServletRequest request)：用于获取用户ID（注意用户ID跟员工ID不是同一个东西。）
* getRoleId(HttpServletRequest request)：获取当前用户的角色ID。
* getLanguage(HttpServletRequest request)：获取当前语言编码

提示：Base Controller中封装的这些方法都是比较常用的方法。在实际项目开发中并不需要自己重新去编写。只要在创建自己的Controller的时候几成BaseController即可。当然如果不满足需求，自己重写一覆盖下就可以了。

### Service层

#### BaseService简介

项目中不管多么复杂的数据库持久层操作，归根结底都是由简单的sql语句组成的。而且不管是那个项目都少不了对单表的增、删、改、查操作。在LCP中，这些基本的表操作不用自己傻傻的每次都写一遍，LCP框架已经帮我们写好了，它就是IBase Service接口定义了一些基本的增删该查。而且该接口是一个泛型接口。如有必要只需在继承该接口并将自己的DTO传入即可。如：public interface IStudentLoginservice extends IBaseService<Student>{}

#### BaseService常用参数类介绍

* IRequest：该类封装了一些从HttpRequest中获取的数据。其常用属性有

String FIELD\_USER\_ID = "userId"; String FIELD\_ROLE\_ID = "roleId";

String FIELD\_LOCALE = "locale"; String EMP\_CODE = "employeeCode";

String FIELD\_LOGIN\_ID="loginId"; String FIELD\_ALL\_ROLE\_ID = "roleIds";

该类中为这些属性提供了get、set方法。如有需要可以在Controller层中将HttpRequest封装进去之后直接调用即可。如：

IRequest requestContext =createRequestContext(request);

#### BaseService常用方法

* List<T> select(IRequest request, T condition, int pageNum, int pageSize)：查询所有满足条件（condition）的结果集。该方法支持分页查询。Request中封装了一些从HttpRequest中的数据。Condition为查询条件，pageNum为当前页码，pageSize为每页显示的记录条数。
* T selectByPrimaryKey(IRequest request, T record)：通过主键查询。Record为对应的DTO。
* List<T> selectAll(IRequest iRequest)：查询全部记录。
* T insert(IRequest request, @StdWho T record)：插入全部数据，如果record中的某个属性为空，该方法会将空插入数据库。
* T insertSelective(IRequest request, @StdWho T record)：插入不为null的数据
* T updateByPrimaryKey(IRequest request, @StdWho T record) 根据主键更新所有数据
* T updateByPrimaryKeySelective(IRequest request, @StdWho T record)：根据主键更新不为null的数据
* List<T> batchUpdate(IRequest request, @StdWho List<T> list)：批量更新。该方法要求T为BaseDTO或者其子类。且默认更新部位null的字段
* int deleteByPrimaryKey(T record)：根据主键删除数据
* int batchDelete(List<T> list)：批量删除

提示:框架的selectAll()、batchDelete()、batchUpdate()这三个方法不支持多语言查询。

### DTO层

#### 常用注解介绍

* @Id：主键Id
* @Column：数据库表字段
* @Transient：暂存字段（在数据当前dto对应的库表中不存在的字段）
* @NotNull：非空验证。用于时间、日期、基本数据类型对应包装类（Long、Integer等）的非空验证
* @NotEmpty：非空验证，用除用于时间、日期、基本数据类型对应包装类之外的数据类型的非空验证
* @MultiLanguageField：多语言字段
* @DateTimeFormat(pattern = "yyyy-MM-dd HH:mm:ss")：日期格式化转换。是Spring框架自带的，用于格式化转换前台传来的日期。绿色部分位具体的日期格式，可根据需要填写
* @JsonFormat(pattern="yyyy-MM-dd HH:mm:ss")：json日期格式转换
* @NumberFormat(pattern="##,###.## ")：数字格式化转换，绿色部分位具体的日期格式，可根据需要填写
* @JsonIgnore：标注序列化时需要忽略的属性。如serialVersionUID等。
* @Table(name = "hss\_task\_orders\_b")：标注当前dto对应的数据库表hss\_task\_orders\_b为数据库表名称。
* @JsonInclude(Include.NON\_NULL)：标注返回的json字符串中是否显示该字段，此处为not\_null，需要显示；若是null则不需要显示。
* @Children：用于投行结构中在头dto中定义对应的行。方便操作头的时候同时操作对应的行。

## 前端开发

适用于kendoUI框架

### 常用组件使用

按照web目录规范，前端文件存放在如下目录

webapp

├─lib

│ ├─bootstrap-3.3.7

│ ├─font-awesome-4.6.3

│ └─kendoui

├─resources

│ ├─css

│ ├─font

│ ├─images

│ ├─js

│ ├─upload

└─WEB-INF

├─web.xml

└─view

资源文件目录 资源文件按照类型划分为两个目录：lib和resources

lib目录存放kendoui的所有文件

resources目录存放程序通用的资源文件

功能文件目录 功能文件存放在view目录下，目录的命名规则按照

[模块代码]/[功能项代码]

例如：用户管理的模块代码是SYS,那么用户管理相关的界面文件都存放在

src/main/webapp/WEB-INF/view/sys下

文件命名约定

界面文件按照如下格式：

[业务对象] [功能操作].html

例如：用户查询界面: user\_query.html

####url命名约定 对于系统中提交的url地址，按照以下格式命名

[contextPath]/[模块代码]/[业务对象]/[操作类型]

对于常见的几种操作类型定义如下

查询URL

[模块代码]/[业务对象]/query

批量提交

[模块代码]/[业务对象]/submit

批量删除

[模块代码]/[业务对象]/remove

例如用户的查询: sys/user/query

####代码编程规范

javascript注释 文件头上定义版权声明以及文件的主要描述信息

/\*\*

\* @summary LCP

\* @description 抽象通用函数

\* @version 1.0

\* @author njq.niu@hand-china.com

\* @copyright Copyright Hand China Co.,Ltd.

\*/

函数上注释

/\*\*

\* 提交表单数据(头行数据可一起提交).

\*

\* <ul>

\* <li>options.form: form对象</li>

\* <li>options.grid: grid对象</li>

\* <li>options.gridName: grid数据的name</li>

\* <li>options.url: 提交的url</li>

\* <li>options.success: success回调函数</li>

\* <li>options.failure: failure回调函数</li>

\* </ul>

\* @param {options}

\*/

freemarker 注释 端界面主要采用freemarker模板技术，文件后缀名统一为.html

<#--

\* Description: 快速编码修改界面

\* Version: 1.0

\* Author: njq.niu@hand-china.com

\* Copyright Hand China Co.,Ltd.

-->

css注释

/\*!

\* description:通用样式文件

\* version: 1.0

\* author: njq.niu@hand-china.com

\* Copyright Hand China Co.,Ltd.

\*/

脚本变量 变量命名原则是减少变量冲突，采用Camel命名法。

var [开发项ID]\_[具体业务含义] = {}

例如：用户管理界面中用户查询的grid数据

var d\_um\_002\_grid = $('#d\_um\_002\_grid').data("kendoGrid");

脚本函数名 函数命名也采用Camel命名法.

function [开发项ID]\_具体业务含义{...}

例如用户查询函数

function d\_um\_002\_query(){

....

}

####开发实践 前端界面采用kendoUI开源框架，相同的显示效果可以采用多种代码方式来实现，为了规范项目组成员的代码样式，统一编码风格，避免以后维护的工作量，根据不同的应用场景总结以下几种开发实践.

通用引入头文件

界面中会引入一些资源文件，例如脚本，样式文件等。为了避免每个界面都重复引入增加工作量以及后续维护的困难，每个界面应该引入通用的文件头。

<#include "../include/header.html">

界面多语言实现

界面中的多语言在数据库中统一存放在sys\_prompts表中，系统启动时加载进redis缓存。界面中实现多语言是通过freemarker的标签以及对应的多语言code来实现。 例如：“确定”的多语言code是“sys.hand.btn.ok”在界面中的写法：

<@spring.message "sys.hand.btn.ok"/>

表单(form)实现

表单主要采用bootstrap的栅格来布局

第一步首先创建一个form的容器对象，一般我们采用div

<div id="d\_um\_002\_form" ></div>

在脚本中根据容器id创建form对象 javascript

window['d\_um\_002\_form'] = $("#d\_um\_002\_form").ligerForm({

fields: [{

type: 'text',

label: 'loginname',

name: 'loginName'

}, {

type: 'text',

label: 'name',

newline: false,

name: 'name'

}],

buttons: [{

text: '查询',

width: 60,

click: function() {

Hap.gridQuery({

form: d\_um\_002\_form,

grid: d\_um\_002\_grid

})

}

}]

});

一般的查询界面都会包含一个查询的表单，以及一个数据展示的grid，查询按钮函数中调用kendoUIGrid的datasource中的read方法去提交查询数据。这里可以直接使用$('#grid').data('kendoGrid').dataSource.page(1);来调用查询方法,查询表单数据一般需要绑定在viewModel中,在查询数据时，将viewModel中的值传递到后台去

viewModel绑定表单数据

1.定义viewModel

var viewModel = kendo.observable({

model : {}

}

2.定义input标签并绑定viewModel

<input type="text" id="promptCode" data-bind="value:model.promptCode">

<script>kendo.bind($('promptCode'), viewModel);</script>

3.查询时传递参数

parameterMap : function(options) {

var map = viewModel.model.toJSON();

map.page = options.page;

map.pagesize = options.pageSize;

return map;

}

表单校验

表单验证采用的是kendoUI提供的kendoValidator组件来验证

<input id="password" type="password"/>

var validator = $("#password").kendoValidator({

messages: {

pwdlen:'<@spring.message "user.password.length"/>'

},

rules: {//自定义规则

pwdlen: function(input){

{

if(input.val().length < 10){

return false;

}

return true;

}

});

对于简单的校验逻辑可直接定义一个验证错误的模板，如必需字段，无需在定义规则。

表单加载数据

加载form数据可以通过页面ajax请求,也可以通过后台传参，在viewModel.model里绑定，可直接加载到表单上

model:{

length: "${length}",

complexity:"${complexity}"

}

表格Grid

定义grid的dom容器

<div style="clear:both">

<div id="grid"></div>

</div>

根据id生成grid对象

$("#grid").kendoGrid({

dataSource:dataSource,

columns : [{

field : "promptCode",

title : '<@spring.message "prompt.promptcode"/>',

width : 240,

headerAttributes: {

"class": "table-header-cell",

style : "text-align: center"

},

attributes : {

//输入只能小写

style : "text-transform:lowercase"

}

},

...

]

});

Grid工具栏

新增按钮

toolbar : [{

name: "create",

template : '<span class="btn btn-primary k-grid-add"><@spring.message "hap.new"/></span>'//自定义模板

}

删除按钮(可在表单自定义按钮)

<span onclick="deleteData()" class="btn btn-danger"><@spring.message "hap.delete"/></span>

function deleteData() {

var checked = grid.selectedDataItems();

if(grid.selectedDataItems().length){

kendo.ui.showConfirmDialog({

title:$l('hap.tip.info'),

message: $l('hap.tip.delete\_confirm')

}).done(function (event) {

if (event.button == 'OK') {

$.each(checked,function(i,v){

grid.dataSource.remove(v)

})

grid.dataSource.sync();

}

})

}

}

保存函数(绑定viewModel方法) ```html <@spring.message "hap.save"/>

```javascript

var viewModel = kendo.observable({

model: {},

saveFunction: function () {

$('#grid').data('kendoGrid').saveChanges();

},

queryResource: function (e) {

$('#grid').data('kendoGrid').dataSource.page(1);

}

});

注意: 为了客户端不再重新刷新，服务端一定要返回提交的数据

return new ResponseData(users);

JSON数据格式

grid和服务端的数据交互统一采用json格式

{

success:true,

rows[{},{}],

total:10

}

success: ajax是否成功的标志 true|false rows: 返回的数据集 total: 总记录数

需要在grid中定义json的格式属性

pageSize: 10, //分页大小

serverPaging: true,//服务器端是否分页

头行关系

头行数据建议采用form+grid的形式.其中form展现头信息，grid展现行信息。 form和grid的定义参考以上章节，这里主要介绍保存逻辑

Hap.submitForm({

url: '${base.contextPath}/sys/lov/submit',

formModel: viewModel.model,

grid: {

lovItems: $('#grid')

},

success: function (result) {

if (!lovId) {

lovId = viewModel.model.lovId

}

}

});

头行保存这里调用了Hap.submitForm方法，参数对象中包含了form对象以及grid对象，gridName定义行数据存放在头上的属性名。

/\*\*

\* 提交表单数据(头行数据可一起提交).

\*

\* <ul>

\* <li>options.formModel: form对象即viewModel.model</li>

\* <li>options.grid: grid对象</li>

\* <li>options.gridName: grid数据的name</li>

\* <li>options.url: 提交的url</li>

\* <li>options.success: success回调函数</li>

\* <li>options.failure: failure回调函数</li>

\* </ul>

\* @param {options}

\*/

通用lov设置

系统中可以设置一些常用的lov方便其他模块调用。 进入系统功能->lov定义

点击新增按钮，创建一个lov

enter image description here

form字段说明：

代码 lov的编码，全局唯一，通过此code获取lov

描述 lov的描述字段，仅用于说明

sqlid lov查询的sqlid，对应后台xml中sql的id

标题 弹出lov的标题

valueField popup中的valueField字段

textField popup中的textField字段

宽度 lov的宽度

高度 lov的高度

grid字段说明

显示 字段的描述(支持多语言)

字段名 字段的name

列宽度 字段的宽度

- 表格列 是否是表格字段

查询字段 是否是查询字段

列序号 表格列排序号

查询配置 如果“查询字段”为“是”，配置查询字段类型

点击"查询配置"，配置查询字段

enter image description here

enter image description here

查询字段可以配置两种类型，文本和下拉框

字段类型 查询字段的类型

查询字段宽度 查询字段宽度

查询字段序号 查询字段排序号

查询字段名 查询字段名

快速编码 当“字段类型”为“下拉框”时指定快速编码的code

url 当“字段类型”为“下拉框”时指定url的url,textFiled,valueFileds

定义好lov配置后，在gird和form中通过以下代码使用lov

grid中

{

field: "parentPositionId",

title: '<@spring.message "position.parentpositionname"/>',

width: 140,

template : function (dataItem) {

return dataItem['parentPositionName'] || '';

},

editor : function (container, options) {

$('<input name="' + options.field + '"/>')

.appendTo(container)

.kendoLov($.extend(<@lov "LOV\_POSITION"/>, {

textField: 'parentPositionName',

query:function(e){

//e.param['positionId'] = options.model.positionId;

},

model : options.model

}));

}

}

form中

<input type="text" id="role" data-bind="value:model.roleId,text:model.roleName">

<script> $("#role").kendoLov(<@lov "LOV\_ROLE"/>)

</script>

字段说明 ：

textField 字段名

model viewModel中的mdel,在grid时，是当前操作绑定的model

query 向lov传递参数

select使用和query相同，在lov选中时触发

dto dto的类名

###多语言编辑器### 框架中提供了TLEdit编辑器来统一多语言设置。 首先更新到最新的框架版本.

form中使用TLEdit

<input id="name" type="text" data-bind="value:model.name">

<script>

$("#name").kendoTLEdit({

idField: 'unitId',

field: 'name',

dto: "com.hand.hap.hr.dto.HrOrgUnit",

model: newViewModel.model

})

kendo.bind($('#name'), newViewModel);

</script>

grid中使用TLEdit

{

field : "functionName",

title : '<@spring.message "function.functionname"/>',

width: 120,

editor: function (container, options) {

$('<input name="' + options.field + '"/>').appendTo(container).kendoTLEdit({

idField: 'functionId',

field: 'functionName',

dto: "com.hand.hap.function.dto.Function",

model: options.model

});

}

}

字段说明 ：

field 多语言字段名

model viewModel中的mdel,在grid时，是当前操作绑定的model

idField dto中主键字段名（确保当前form中有对应的值)

dto dto的类名

### 常用函数介绍

#### 表单提交

表单提交可以采用LCP框架封装的submitForm(options)方法。options中需要3个属性，分别是提交路径url、所需提交的formModel、回调函数（如：success()）。例：

Hap. submitForm({

url: '${base.contextPath}/elp/expressManage/headerSubmit',

formModel: viewModel.model,

success: function (json){ //此处做回调处理。比如提示成功、失败等等}

})

#### grid删除

删除操作可以调用hap封装的deleteGridSelection方法。例：

Hap.deleteGridSelection({

grid:$('#express\_manage\_grid')

});

#### 参数处理

prepareSubmitParameter(options,type)：提交参数处理

prepareQueryParameter (options,type)：查询参数处理

这两个方法也是LCP封装的，可直接调用。如在grid的数据源中配置parameter Map：

var dataSource = new kendo.data.DataSource({

transport : {

…….

parameterMap: function (options, type) {

if (type !== "read" && options.models) {

return kendo.stringify (Hap.prepareSubmitParameter(options, type))

} else if (type === "read") {

return Hap.prepareQueryParameter(viewModel.model.toJSON(), options)

}

}

…….

}

});

#### Grid操作

Grid查询用的是kendo UI 框架封装的方法：

grid查询：datasource的page()方法

grid修改：saveChanges()

新建：addRow()

删除：一般根据需要采用自定的js方法

## 入门案例

### 实例介绍

本实例基于LCP3.0框架实现了一个简单的CRUD功能的项目的开发，通过介绍本次项目的开发流程，使初学者初步认识LCP3.0的开发

先睹为快：



### 后台开发



#### 数据库

本次开发数据库选用的是 MySql，测试数据如下:

|  |
| --- |
| SET FOREIGN\_KEY\_CHECKS=0;  -- ----------------------------  -- Table structure for my\_test01  -- ----------------------------  DROP TABLE IF EXISTS `my\_test01`;  CREATE TABLE `my\_test01` (  `MY\_ID` varchar(64) COLLATE utf8\_bin NOT NULL,  `MY\_NAME` varchar(64) COLLATE utf8\_bin DEFAULT NULL,  `MY\_AGE` int(12) DEFAULT NULL,  `MY\_BIRTHDAY` timestamp NULL DEFAULT NULL,  PRIMARY KEY (`MY\_ID`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8\_bin;  -- ----------------------------  -- Records of my\_test01  -- ----------------------------  INSERT INTO `my\_test01` VALUES ('1', '小明', '20', '2016-10-13 00:00:00');  INSERT INTO `my\_test01` VALUES ('2', '静静', '30', '2016-09-08 00:00:00');  INSERT INTO `my\_test01` VALUES ('3', '傻龙', '20', '2016-10-20 00:00:00');  INSERT INTO `my\_test01` VALUES ('4', '大壮', '40', '2016-10-01 00:00:00'); |

#### Mapper层

LCP3.0后台部分和之前版本基本没有改变，这里就不再赘述。在完成 Dto类的封装后在Mapper层中定义基本的CRUD函数。

|  |
| --- |
| public interface DemoMapper extends Mapper<Demo> {  List<Demo> selectByDemo(Demo demo);  int insertDemo(Demo demo);  int updateDemo(Demo demo);  int deleteDemo(Demo demo);  } |

然后在对应的 Mapper.xml 文件中配置对应的SQL语句。

|  |
| --- |
| <?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="hbi.core.demo.mapper.DemoMapper">  <resultMap id="BaseResultMap" type="hbi.core.demo.dto.Demo" extends="com.hand.hap.mapper.StdMapper.STD">  <result column="MY\_ID" property="myId" jdbcType="VARCHAR"/>  <result column="MY\_NAME" property="myName" jdbcType="VARCHAR"/>  <result column="MY\_AGE" property="myAge" jdbcType="DECIMAL"/>  <result column="MY\_BIRTHDAY" property="myBirthday" jdbcType="TIMESTAMP"/>  </resultMap>  <select id="selectByDemo" parameterType="hbi.core.demo.dto.Demo" resultMap="BaseResultMap">  SELECT m.MY\_ID,  m.MY\_NAME,  m.MY\_AGE,  m.MY\_BIRTHDAY  FROM my\_test01 m  <where>  <if test="myId != null">  and m.MY\_ID like #{myId,jdbcType=VARCHAR}  </if>  <if test="myName != null">  and m.MY\_NAME like #{myName,jdbcType=VARCHAR}  </if>  <if test="myAge != null">  and m.MY\_AGE = #{myAge,jdbcType=DECIMAL}  </if>  <if test="myBirthday != null">  and m.MY\_BIRTHDAY = #{myBirthday,jdbcType=TIMESTAMP}  </if>  </where>  </select>  <insert id="insertDemo" parameterType="hbi.core.demo.dto.Demo">  INSERT INTO my\_test01  <trim prefix="(" suffix=")" suffixOverrides="," >  <if test="myId != null" >  MY\_ID,  </if>  <if test="myName != null" >  MY\_NAME,  </if>  <if test="myAge != null">  MY\_AGE,  </if>  <if test="myBirthday != null">  MY\_BIRTHDAY,  </if>  </trim>  <trim prefix="values (" suffix=")" suffixOverrides="," >  <if test="myId != null" >  #{myId,jdbcType=VARCHAR},  </if>  <if test="myName != null" >  #{myName,jdbcType=VARCHAR},  </if>  <if test="myAge != null">  #{myAge,jdbcType=DECIMAL},  </if>  <if test="myBirthday != null">  #{myBirthday,jdbcType=TIMESTAMP},  </if>  </trim>  </insert>  <update id="updateDemo" parameterType="hbi.core.demo.dto.Demo" >  update my\_test01  <set >  <if test="myId != null" >  MY\_ID = #{myId,jdbcType=VARCHAR},  </if>  <if test="myName != null" >  MY\_NAME = #{myName,jdbcType=VARCHAR},  </if>  <if test="myAge != null">  MY\_AGE = #{myAge,jdbcType=DECIMAL},  </if>  <if test="myBirthday != null">  MY\_BIRTHDAY = #{myBirthday,jdbcType=TIMESTAMP},  </if>  </set>  where MY\_ID = #{myId,jdbcType=VARCHAR}  </update>  <delete id="deleteDemo" parameterType="hbi.core.demo.dto.Demo" >  delete from my\_test01  where MY\_ID= #{myId,jdbcType=DECIMAL}  </delete>  </mapper> |

#### Service层

首先封装 Service 层的接口类，项目中的查询通过 selectByDemo() 实现，新增和更改通过 batchUpdate() 实现，删除通过框架中封装的 batchDelete() 实现。

|  |
| --- |
| public interface IDemoService extends IBaseService<Demo>, ProxySelf<IDemoService> {  List<Demo> selectByDemo(IRequest requestContext, Demo demo, int page, int  pagesize);  List<Demo> batchUpdate(IRequest requestContext, @StdWho List<Demo>  demos);  } |

然后是封装 Service 层的实现类。在 batchUpdate() 中通过判断 Dto 对象的状态来选择新增或者更改的情况。

|  |
| --- |
| @Service  @Transactional  public class DemoServiceImpl extends BaseServiceImpl<Demo> implements IDemoService {  @Autowired  private DemoMapper demoMapper;    @Override  @Transactional(propagation = Propagation.SUPPORTS, rollbackFor =  Exception.class)  public List<Demo> selectByDemo(IRequest requestContext, Demo demo, int  page, int pagesize) {  PageHelper.startPage(page, pagesize);  return demoMapper.selectByDemo(demo);  }    @Override  @Transactional(propagation = Propagation.SUPPORTS)  public List<Demo> batchUpdate(IRequest requestContext, List<Demo> demos){  for (Demo demo : demos) {  if (demo.get\_\_status() != null) {  switch (demo.get\_\_status()) {  case DTOStatus.ADD:  demoMapper.insertDemo(demo);  break;  case DTOStatus.UPDATE:  demoMapper.updateDemo(demo);  break;  default:  break;  }  }  }  return demos;  }  } |

#### Controller层

这里提供了三个 url 分别供查询，增改和删除时调用。其中 batchDelete() 是 IDemoService 继承自 IBaseService 的方法。

|  |
| --- |
| @Controller  public class DemoController extends BaseController{  @Autowired  private IDemoService demoService;  @RequestMapping("/demo/query")  @ResponseBody  public ResponseData selectList(HttpServletRequest request, Demo condition,  @RequestParam(defaultValue = DEFAULT\_PAGE) int page,  @RequestParam(defaultValue = DEFAULT\_PAGE\_SIZE) int pagesize) {  IRequest iRequest = createRequestContext(request);  List<Demo> datas = demoService.selectByDemo(iRequest, condition, page,  pagesize);  return new ResponseData(datas);  }  @RequestMapping(value = "/demo/submit", method = RequestMethod.POST)  @ResponseBody  public ResponseData submit(HttpServletRequest request, @RequestBody  List<Demo> demos) {  IRequest iRequest = createRequestContext(request);  List<Demo> datas = demoService.batchUpdate(iRequest, demos);  return new ResponseData(datas);  }  @RequestMapping(value = "/demo/delete", method = RequestMethod.POST)  @ResponseBody  public ResponseData delete(HttpServletRequest request, @RequestBody  List<Demo> demos) {  IRequest iRequest = createRequestContext(request);  demoService.batchDelete(demos);  return new ResponseData();  }  } |

### 前台开发

#### MVVM

KendoUI没有封装Form组件，LCP3.0中通过KendoUI的MVVM实现组件和数据源的无缝结合，进而实现Form表单。

MVVM即Model View View-Model，是一种设计模式，用来对Model层和View层进行分层，View-Model负责暴露数据层和视图层交互的数据。

首先，创建一个View-Model对象，其中的Model根据你的数据产生。我们在这里定义供组件调用的函数。

|  |
| --- |
| var viewModel = kendo.observable({  model: {},  createFunction: function(){  //$('#grid').data('kendoGrid').addRow();  editData();  },  saveFunction: function(){  $('#grid').data('kendoGrid').saveChanges();  },  queryFunction: function (e) {  $('#grid').data('kendoGrid').dataSource.page(1);  },  resetForm : function(e) {  var formData = viewModel.model.toJSON();  for ( var k in formData) {  viewModel.model.set(k, null);  }  }  }); |

然后，创建 View 层，由 html 元素组成，用来绑定 View-Model。在 LCP3.0 中在这里定义表单，由 KendoUI 的组件和 Bootstrop 的栅格布局组成。

|  |
| --- |
| <div id="page-content">  <div class="pull-left" id="toolbar-btn" style="padding-bottom:10px;">  <span class="btn btn-primary k-grid-add"  style="float:left;margin-right:5px;" data-bind="click:createFunction">  <@spring.message "hap.new"/></span>  <span class="btn btn-success k-grid-save-changes"  style="float:left;margin-right:5px;" data-bind="click:saveFunction">  <@spring.message "hap.save"/></span>  <span onclick="deleteData()" class="btn btn-danger" style="float:left;">  <@spring.message "hap.delete"/></span>  </div>    <div class="pull-right" id="query-form" style="padding-bottom:10px;">  <input id="myId" data-role="maskedtextbox" type="text"  placeholder='<@spring.message "demo.myId"/>'  style="float:left;width:150px;margin-right:5px;"  data-bind="value:model.myId" class="k-textbox">  <input type="text" id="myName" placeholder='<@spring.message  "demo.myName"/>' data-bind="value:model.myId,  text:model.myName">  <select id="myAge" data-value-primitive="true"  placeholder='<@spring.message "demo.myAge"/>'  data-bind="value: model.myAge"></select>  <input type="date" id="myBirthday" placeholder='<@spring.message  "demo.myBirthday"/>' data-bind="value:model.myBirthday">  <div style="margin:10px;margin-right:0px;">  <div class="col-md-9"></div>  <span class="btn btn-default" style="float:right;width:70px"  data-bind="click:resetForm" type="button"><@spring.message  "hap.reset"/></span>  <span class="btn btn-primary" style="float:right;width:70px;  margin-right:5px;" data-bind="click:queryFunction"  type="submit"><@spring.message "hap.query"/></span>  </div>  <div style="clear:both"></div>  </div>  <div style="clear:both">  <div id="grid"></div>  </div>  </div> |

从代码里，我们可以看到 input 元素的 value 属性跟 View-Model 的字段绑定。于是，当 View-Model 的字段的值发生改变时，input 元素的value值也随着改变。反之，input 元素的 value 值发生改变，View-Model 的字段的值也随着发生变化。而且变化是实时的。

我们还看到，button 元素的 click 事件与 View-Model 的方法发生绑定。于是，当你点击 button 时，便调用该方法。

最后，我们把 View 和 View-Model 进行绑定。

|  |
| --- |
| kendo.bind($('#page-content'), viewModel); |

绑定格式 <binding name> : <view model field or method> ，你可以绑定 html 元素的其他属性，source，html，attr，visible，enabled 等等。你也可以连续绑定多个 html元 素，比如 data-bind="value: name, visible: isNameVisible"。或者如下这样绑定，这里是独立的例子，与本项目无关。

|  |
| --- |
| <div data-bind="text: person.name"></div>  <script>       var viewModel = kendo.observable({           person: {               name: "John Doe"           }       });       kendo.bind($("div"), viewModel);  </script> |

#### Datasource

KendoUI 的数据源支持本地数据源（JavaScript 对象数组），或者远程数据源（XML, JSON, JSONP)。支持 CRUD 操作，并支持排序，分页，过滤，分组和集合等。

在 LCP3.0 中，我们使用后台提供的三个url作为远程数据源，从而获得到可以提供给表单和表格的数据。

|  |
| --- |
| var crudServiceBaseUrl = '${base.contextPath}',  dataSource = new kendo.data.DataSource({  transport : {  read : {  url : crudServiceBaseUrl + "/demo/query",  type : "POST",  dataType: "json"  },  create : {  url : crudServiceBaseUrl + "/demo/submit",  contentType: "application/json",  type : "POST"  },  update: {  url: crudServiceBaseUrl + "/demo/submit",  contentType: "application/json",  type: "POST"  },  destroy: {  url: crudServiceBaseUrl + "/demo/delete",  contentType: "application/json",  type: "POST"  },  parameterMap: function (options, type) {  if (type !== "read" && options.models) {  var datas = Hap.prepareSubmitParameter(options, type);  return kendo.stringify(datas);  } else if (type === "read") {  return Hap.prepareQueryParameter  (viewModel.model.toJSON(), options);  }  }  },  batch : true,  serverPaging: true,  serverSorting: true,  pageSize : 10,  schema : {  data : 'rows',  total : 'total',  model : {  id : "myId",  fields: { myId : {type: "string"},myName : {type: "string"},  myAge : {type: "long"},myBirthday : {type: "date"}  }  }  }  }); |

#### 表格

KendoUI 有封装好的 grid 组件，LCP3.0中我们在 View 层定义一个子标签用来承载 grid 组件。所以表格的数据 dataSource 也是 View 层的数据，在 View 层的其他子标签也共用这个数据，如 form，toolbar 等。

这里使用了 MVVM 的原理，将 View 层和数据源结合。这里的逻辑比较复杂，我会针对在LCP3.0中的使用情况将我的理解用形象易懂的语言介绍一下。

MVVM 分为三个部分，Model 数据层，View 视图层和 View-Model对象。在 LCP2.0 中即上面介绍的 dataSource ，<div id="page-content"> 和 viewModel。

dataSource 是我们从后台获得的数据。<div id="page-content"> 是一个父级标签， toolbar 按钮组，form 表单和 grid 表格等都是其子标签，他们共用同一数据。 viewModel 用于实现数据层和视图层的交互，其中的 model 根据数据产生。

|  |
| --- |
| var grid = $("#grid").kendoGrid({  dataSource : dataSource,  navigatable: false,  height: '300',  weight: '180',  resizable: true,  scrollable: true,  selectable:"multiple,rowbox",  pageable: {  pageSizes: [5, 10, 20, 50],  refresh: true,  buttonCount: 5  },  sortable: true,  columns : [  {  field: "myId",  title: '<@spring.message "demo.myId"/>',  attributes: {style: "text-align:center"},  headerAttributes: {  "class": "table-header-cell",  style : "text-align: center"  },  width: 80  },  {  field: "myName",  title: '<@spring.message "demo.myName"/>',  attributes: {style: "text-align:center"},  headerAttributes: {  "class": "table-header-cell",  style : "text-align: center"  },  width: 80  },  {  field: "myBirthday",  attributes: {style: "text-align:center"},  title: '<@spring.message "demo.myBirthday"/>',  width: 80,  headerAttributes: {  "class": "table-header-cell",  style : "text-align: center"  },  format: "{0:yyyy-MM-dd}"  },  {  field : "myAge",  title : '<@spring.message "demo.myAge"/>',  attributes: {style: "text-align:center"},  headerAttributes: {  "class": "table-header-cell",  style : "text-align: center"  },  width : 80  },  { //编辑  title: '<@spring.message "hap.edit"/>',  width: 50,  template: function (rowdata) {  if (!!rowdata.myId) {  return '<a href="javascript:void(0);"  onclick="editFunctionResources('+rowdata.myId +')  "><@spring.message "hap.edit"/></a>'  }  return '';  },  attributes: {style: "text-align:center"},  headerAttributes: { style: "text-align:center"}  }  ],  editable : true  }).data("kendoGrid");    //自动根据当前屏幕大小调整表格  Hap.autoResizeGrid("#grid"); |

#### 文本框

KendoUI 中没有表单 form 组件，在LCP3.0 中通过使用 Html5 的标签和 BootStrap 的栅格布局实现。即 <div id="query-form">，是 <div id="page-content">

的子标签。而文本框是 <div id="query-form"> 的子标签。

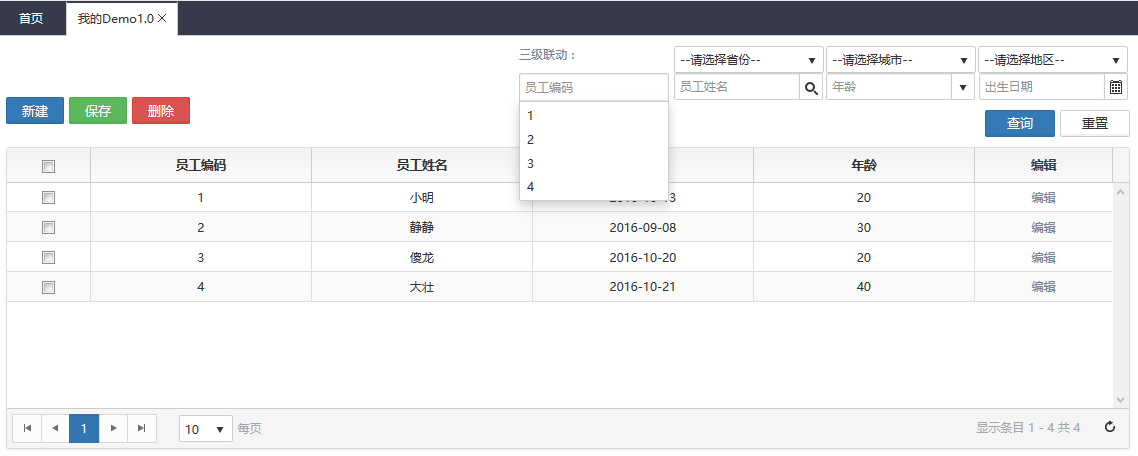
|  |
| --- |
| <div class="form-group">  <label class="col-sm-2 control-label">文本框标题</label>  <div class="col-sm-10">  <input type="text" data-role="maskedtextbox" placeholder='文本框填充'  style="" data-bind="value:model.myId" class="k-textbox" id="myId">  </div>  </div> |

KendoUI 中的文本框通过 data-role="maskedtextbox" 和 class="k-textbox" 实现样式，通过 data-bind="value:model.myId" 实现绑定数据。

当 KendoUI 的组件被加载时，首先通过 data-role 判断当前组件是什么，如果 data-role 不存在会默认为普通文本框，然后根据 data-bind 绑定数据。

|  |
| --- |
| var myLocalDataSource = new kendo.data.DataSource({data:myData});  $("#myId").kendoAutoComplete({  dataTextField:"myId",  dataSource:myLocalDataSource  }); |

KendoUI 中可以通过 kendoAutoComplete() 为文本框添加下拉框，其中myData 是一个数组，通过 new kendo.data.DataSource() 变成可以引用的数据。

如图所示: 

#### 下拉框

KendoUI 中可以通过 kendoComboBox() 和 kendoDropDownList() 实现两种效果的下拉框，后者在介绍级联时使用并介绍。

|  |
| --- |
| <select id="myAge" data-value-primitive="true" data-bind="value: model.myAge">  </select>  $("#myAge").kendoComboBox({  dataTextField:"myAge",  dataValueField:"myAge",  animation: {  close: {  effects: "fadeOut zoom:out",  duration: 300  },  open: {  effects: "fadeIn zoom:in",  duration: 300  }  },  dataSource:myLocalDataSource  }); |

其中 data-value-primitive 绑定会是一条数据而不是一个对象，dataTextField 指定显示字段，dataValueField 指定显示属性字段，animation 定义下拉框展开和关闭时的动画效果。

如图所示: 

#### 时间选择器

KendoUI中可以通过id将kendoDatePicker()和标签绑定实现日历。

|  |
| --- |
| <input id="myBirthday" type="date" data-bind="value:model.myBirthday">  $("#myBirthday").kendoDatePicker({  animation: {  close: {  },  open: {  }  },  format: "{0:yyyy-MM-dd}",  change: function(){  }  }); |

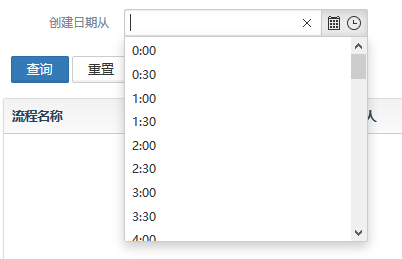
其中 format 定义时间的格式，change 定义当时间改变时触发的事件。

如图所示: 

KendoUI 中可以通过 data-role="datetimepicker" 将标签实现为日历和时间。

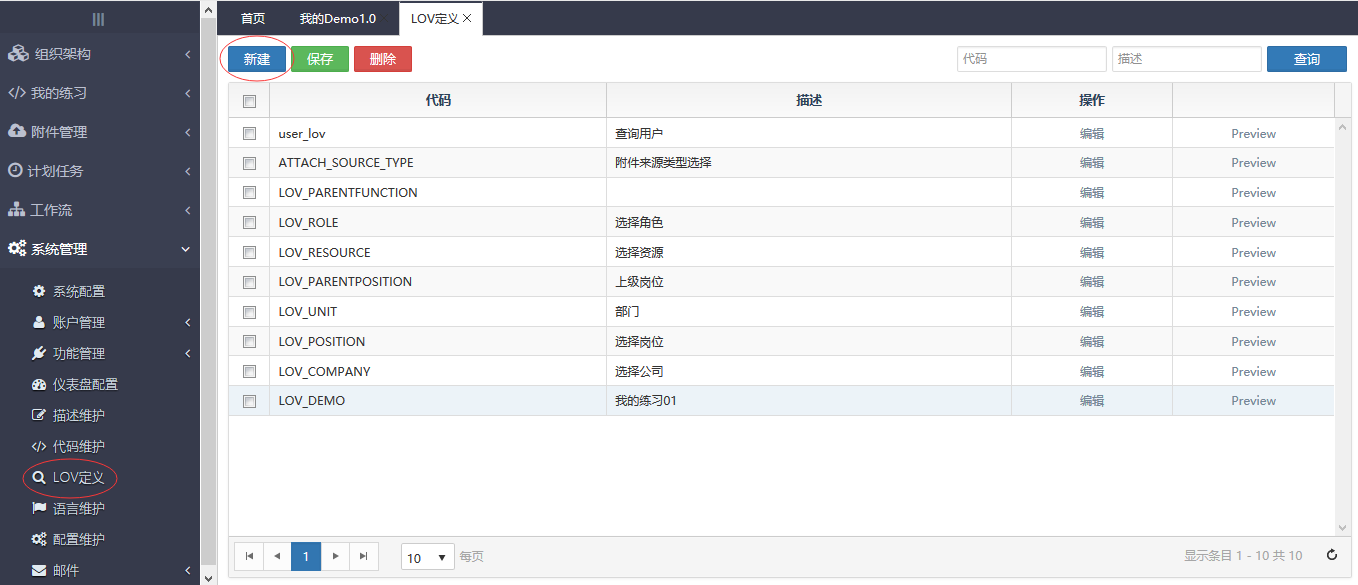
|  |
| --- |
| <input data-role="datetimepicker" data-bind="value:model.myBirthday" class="k-datetimepicker"> |

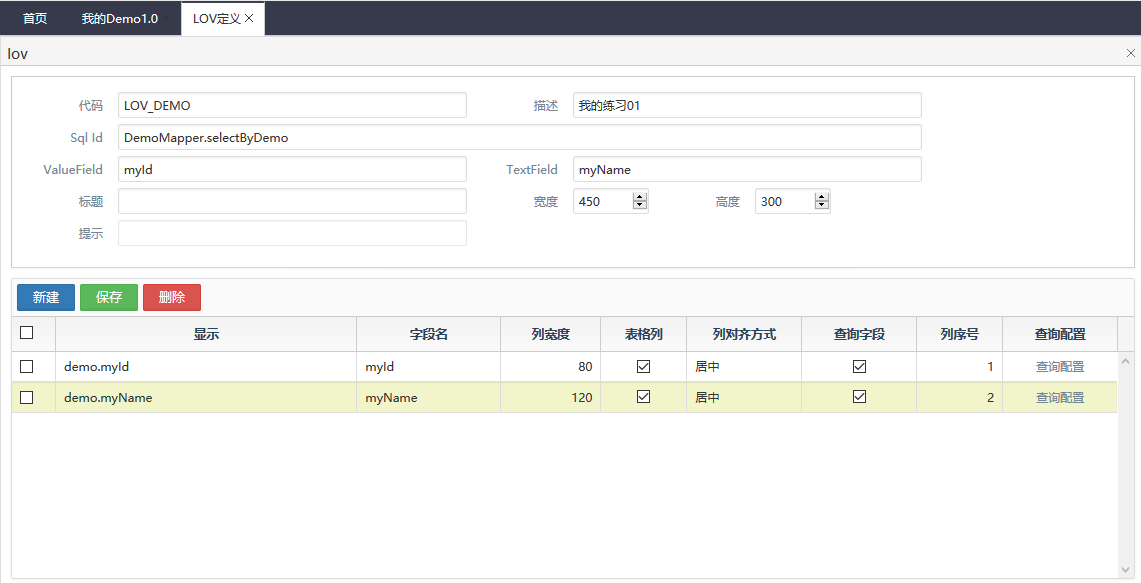
如图所示:



#### LOV

在LCP3.0框架中实现 LOV 首先要自定义一个 LOV。以下是具体步骤。

如图所示: 然后在新建页面中进行编辑。

如图所示: 

其中 Sql Id是Mapper.xml中Sql语句的id。

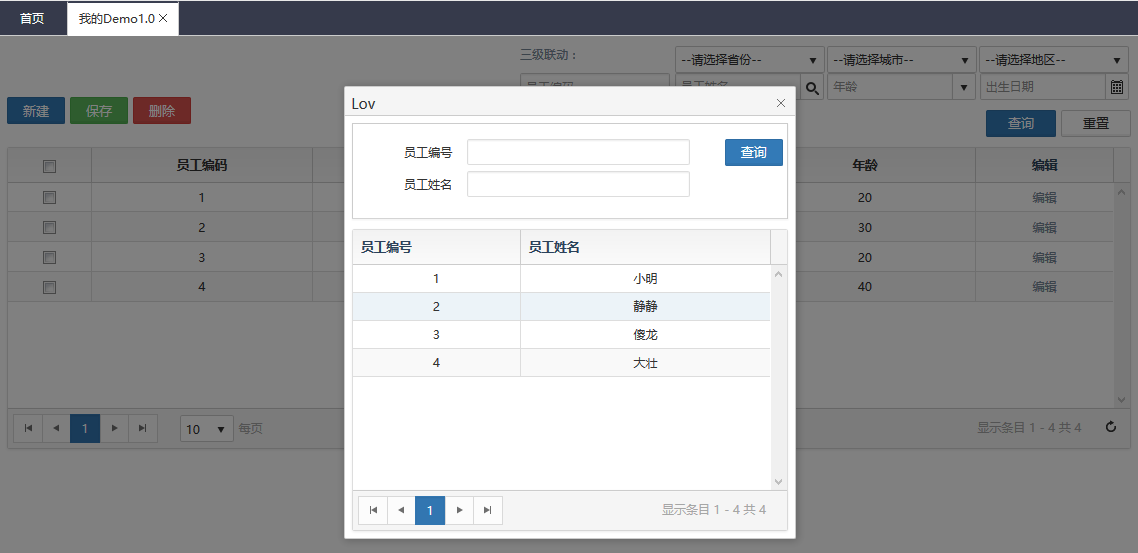
|  |
| --- |
| <select id="selectByDemo" parameterType="hbi.core.demo.dto.Demo"  resultMap="BaseResultMap"> |

这样一个叫 LOV\_DEMO 的 LOV 就定义好了，并且可以在代码中直接调用。

以下是在代码中调用 LOV 的具体步骤。

|  |
| --- |
| <input type="text" id="myName" data-bind="value:model.myId,  text:model.myName">  $("#myName").kendoLov(${lovProvider.getLov(base.contextPath, base.locale,  "LOV\_DEMO")}) |

KendoUI 并没有实现 LOV，所以没有 data-role，LCP3.0 中通过 kendoLov() 将标签变成 LOV。其中 data-bind 中 text 的值从 LOV 中选中数据后会在输入框中显示的字段。

如图所示: 

#### 查询与重置

这里的标签是 <div id="query-form"> 的子标签。Click 事件调用的函数已经在 viewModel 中封装完成。

|  |
| --- |
| <div>  <span class="btn btn-default" data-bind="click:resetForm" type="button">  <@spring.message "hap.reset"/></span>  <span class="btn btn-primary" data-bind="click:queryFunction" type="submit">  <@spring.message "hap.query"/></span>  </div> |

其中 keydown() 是表单回车查询，query-form 是表单的 id，input 指表单内所有输入框，queryResource(e) 是当按下回车按钮时执行的查询方法 。

|  |
| --- |
| var viewModel = kendo.observable({  model: {},  queryFunction: function (e) {  $('#grid').data('kendoGrid').dataSource.page(1);  },  resetForm : function(e) {  var formData = viewModel.model.toJSON();  for ( var k in formData) {  viewModel.model.set(k, null);  }  }  });    kendo.bind($('#page-content'), viewModel);    $('#query-form input').keydown(function (e) {  if (e.keyCode == 13) {  e.target.blur();  viewModel.queryFunction(e);  }  }); |

#### Toolbar

这里的标签是 <div id="page-content"> 的子标签。新建和保存的 Click 事件调用的函数已经在 viewModel 中封装完成，deleteData() 在 viewModel 外进行封装。

|  |
| --- |
| <div class="pull-left" id="toolbar-btn">  <span class="btn btn-primary k-grid-add" data-bind="click:createFunction">  <@spring.message "hap.new"/></span>  <span class="btn btn-success k-grid-save-changes"  data-bind="click:saveFunction"><@spring.message "hap.save"/></span>  <span onclick="deleteData()" class="btn btn-danger"><@spring.message  "hap.delete"/></span>  </div> |

这里介绍的新建是在表格中增加一行进行新建，dialog 新建会在之后的新建和编辑中介绍。

|  |
| --- |
| var viewModel = kendo.observable({  model: {},  createFunction: function(){  $('#grid').data('kendoGrid').addRow();  },  saveFunction: function(){  $('#grid').data('kendoGrid').saveChanges();  }  });    kendo.bind($('#page-content'), viewModel);  function deleteData() {  var checked = grid.selectedDataItems();  if(grid.selectedDataItems().length){  kendo.ui.showConfirmDialog({  title:$l('hap.tip.info'),  message: $l('hap.tip.delete\_confirm')  }).done(function (event) {  if (event.button == 'OK') {  $.each(checked,function(i,v){  grid.dataSource.remove(v)  })  grid.dataSource.sync();  }  })  }  } |

#### 新建与编辑1

我将新建与编辑分为两个部分介绍，第一部分是原始页面部分。

我们在自定义新建和编辑的弹窗时，首先需要一个标签来承载这个 window。该标签写在 <div id="page-content"> 外即可。

|  |
| --- |
| <div id="dialog"></div> |

然后封装 editData 函数，作为 Click 事件调用的最后一级函数。这里的 url 与编辑的 url 相同，在跳转到该页面后再进行判断是新建还是编辑。

|  |
| --- |
| <span class="btn btn-primary k-grid-add" data-bind="click:createFunction">  <@spring.message "hap.new"/>  </span>  var viewModel = kendo.observable({  model: {},  createFunction: function(){  editData();  }  });    kendo.bind($('#page-content'), viewModel);  //新建时调用的界面  window.editData = function(){  var url = 'demo\_edit.html';  var dialog = $("#dialog").kendoWindow({  actions: [ "Maximize", "Minimize", "Close"],  width: 900,  height: 500,  title: '<@spring.message "hap.edit"/>',  content: url,  iframe: true,  visible: false,  modal:true,  close: function() {  //window 关闭 刷新 本页面的 grid  $('#grid').data('kendoGrid').dataSource.page(1);  }  }).data("kendoWindow");  dialog.center().open();  }; |

这里的编辑作为 grid 的一个列元素，首先封装 editFunctionResources 函数供 onclick 事件调用。由于编辑需要原始内容，所以将 myId 作为参数传入，这也是跳转页面判断新建还是编辑的依据。

|  |
| --- |
| //编辑  {  title: '<@spring.message "hap.edit"/>',  width: 50,  template: function (rowdata) {  if (!!rowdata.myId) {  return '<a href="javascript:void(0);" onclick=  "editFunctionResources('+rowdata.myId +')">  <@spring.message "hap.edit"/></a>'  }  return ' ';  },  attributes: {style: "text-align:center"},  headerAttributes: { style: "text-align:center"}  }  //编辑弹窗函数  editFunctionResources = function (myId) {  var dialog = $("#dialog").kendoWindow({  actions: ["Close"],  width : 900,  height : 500,  title : '<@spring.message "hap.edit"/>',  visible: false,  iframe : true,  modal : true,  content: 'demo\_edit.html?myId=' + myId,  close:function(e){  $("#grid").data("kendoGrid").dataSource.page(1);  }  }).data("kendoWindow");  dialog.center().open();  }; |

#### 新建与编辑2

第二部分是跳转页面部分。首先进行是否传入参数的判断，这里决定着两个方面。第一方面是是否加载原始数据，第二方面是在判断新建还是更新。

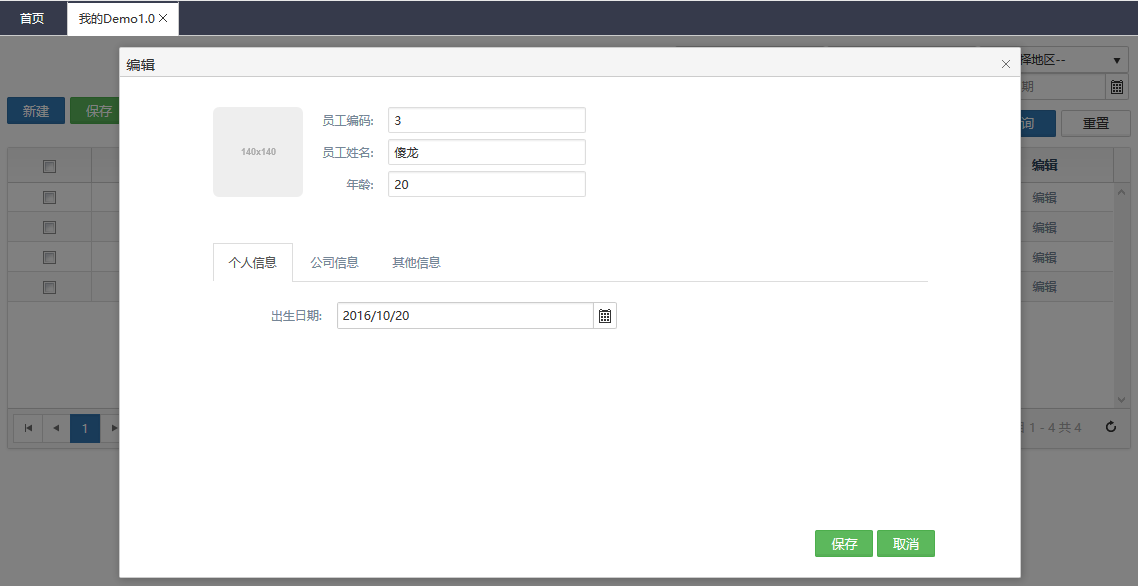
|  |
| --- |
| var isEdit = false;  var myId = '${RequestParameters.myId!0}';  if (myId!=0) {  isEdit = true;  $.ajax({  url: '${base.contextPath}/demo/query?myId=  ${RequestParameters.myId!0}',  success: function (args) {  var a0 = args.rows[0] || {};  for (var k in a0) {  viewModel.model.set(k, a0[k]);  }  }  });  } |

然后是对数据的编辑和保存，这里还是利用 MVVM 设计模式。具体请参考以上内容。首先定义 View-Model 对象。

|  |
| --- |
| var viewModel = kendo.observable({  model: {},  save: function(e){  var data= viewModel.model.toJSON();  if(isEdit){  data.\_\_status = "update";  }else{  data.\_\_status = "add";  }  var validator = $("#mainform").data("kendoValidator");  if (validator.validate()) {  $.ajax({  type : 'POST',  url : '${base.contextPath}/demo/submit',  dataType : "json",  contentType : "application/json",  data : kendo.stringify([data]),  success: function (data) {  if(data.success==false){  kendo.ui.showErrorDialog({  message:data.message  });  }else{  window.parent.$("#dialog").data("kendoWindow").close();  }  }  });  }},  closeWin: function(e){  window.parent.$("#dialog").data("kendoWindow").close();  }}); |

然后定义 View 层。同样是通过使用 Html5 的标签和 BootStrap 的栅格布局实现。在这里给出基本架构的代码，tab 标签页部分在接下来的部分介绍。

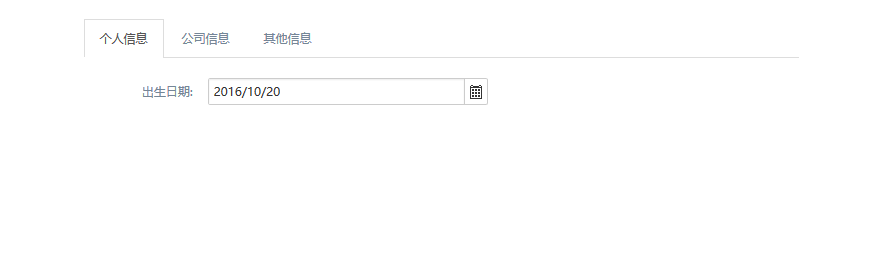
|  |
| --- |
| <div id="page-content">  <form id="mainform" class="form-horizontal" method="post"  enctype="application/json;charset=UTF-8">  <div class="row">  <div class="form-group">  <div class="col-sm-11" style="float: left;margin-top: 0px;">  <div class="form-group">  <label class="col-sm-2 control-label"><@spring.message  "demo.myId"/>:</label>  <div class="col-sm-4">  <input type="text" required style="width:100%" name="myId"  data-bind="value:model.myId" class="k-textbox">  </div>  <div class="col-sm-6" >  <span data-for="myId" class=".k-invalid-msg"></span>  </div>  </div>  </div>  </div>  </div>  <div class="text-right" style="bottom: 20px;position: fixed; right: 85px; float: left;  background: #fff;">  <span class="btn btn-success" data-bind="click:save" type="submit">  <@spring.message "hap.save"/></span>  <span class="btn btn-success" data-bind="click:closeWin" type="button">  <@spring.message "hap.cancel"/></span>  </div>  </form>  </div> |

如图所示: 

#### Tab

Tab 标签页同样是通过使用 Html5 的标签和 BootStrap 的栅格布局实现。

|  |
| --- |
| <div class="col-sm-12" style="margin-top: 40px;">  <ul class="nav nav-tabs" id="mytab">  <li class="active"><a href="#personal" data-toggle="tab">  <@spring.message "employee.personalinfo"/></a></li>  <li class=""><a href="#company" data-toggle="tab"><@spring.message  "employee.companyinfo"/></a></li>  <li class=""><a href="#another" data-toggle="tab"><@spring.message  "employee.otherinfo"/></a></li>  </ul>  <div id="myTabContent" class="tab-content">  <div class="tab-pane fade in active" style="margin-top: 20px;"  id="personal">  <div class="form-group">  <label class="col-sm-2 control-label"><@spring.message  "demo.myBirthday"/>:</label>  <div class="col-sm-5">  <input style="width:100%" name="myBirthday" data-bind  ="value:model.myBirthday" class="datepicker">  </div>  <div class="col-sm-5" >  <span data-for="myBirthday" class=".k-invalid-msg">  </span>  </div>  </div>  </div>  <div class="tab-pane fade" style="margin-top: 20px;" id="company">  <div style="margin-top: 20px; col-sm-2">  <p><@spring.message "employee.companyinfo"/></p>  </div>  </div>  <div class="tab-pane fade" id="another">  <div style="margin-top: 20px; col-sm-2">  <p><@spring.message "employee.otherinfo"/></p>  </div>  </div>  </div>  </div> |

如图所示: 

#### 三级联动

本次开发中的三级联动是独立模块，表结构和后台设计都是独立的。后台只需提供一个基本查询的 url。

本次实现的是中国的省，市和县的三级联动。下面是使用的表结构。



在 LCP3.0 中实现级联，首先我们要得到数据源，即下拉框中的内容。这里我将需要的字段放入数组中，方便之后调用。

|  |
| --- |
| var myData = [];  function myDataToJson(data) {  this.regionName = data.regionName;  this.regionId = data.regionId;  this.parentId = data.parentId;  }    $.ajax({  url : '${base.contextPath}/ass/region/query',  type : 'POST',  contentType : "application/json;charset=utf-8",  cache : false,  dataType : 'json',  async : false,  success : function(data) {  for (var i = 1; i < data.rows.length; i++) {  myData.push(new myDataToJson(data.rows[i]));  }  }  }); |

然后初始化控件，使三个下拉框都使用myData作为数据。

|  |
| --- |
| <label style="float:left;width:150px;margin-right:5px;">三级联动：</label>  <select id="first" style="float:left;width:150px;margin-right:2px;"></select>  <select id="second" style="float:left;width:150px;margin-right:2px;"></select>  <select id="third" style="float:left;width:150px;margin-right:2px;"></select>  $(document).ready(function(){  //初始化控件  firstDropDownList = $("#first").kendoDropDownList({  optionLabel: " --请选择省份--",  dataTextField:"regionName",  dataValueField:"regionName",  dataSource:{  data:myData  },  change:function(){  }  }).data("kendoDropDownList");    secondDropDownList = $("#second").kendoDropDownList({  optionLabel: " --请选择城市--",  dataTextField:"regionName",  dataValueField:"regionName",  dataSource:{  data:myData  },  change:function(){  }  }).data("kendoDropDownList");    thirdDropDownList = $("#third").kendoDropDownList({  optionLabel: " --请选择地区--",  dataTextField:"regionName",  dataValueField:"regionName",  dataSource:{  data:myData  }  }).data("kendoDropDownList");    //初始数据过滤器  }) |

接下来是初始化过滤器。在设置过滤器之前，省市县的数据都是中国的所有地区，通过过滤器来筛选数据。

|  |
| --- |
| //省 firstDropDownList.dataSource.filter({ field: "parentId", operator: "eq", value: 1 });  //市 var one\_filter={field:"parentId", operator:"eq", value: parseInt(firstDropDownList.dat aItem().regionId)};    secondDropDownList.dataSource.filter(one\_filter);  //县 var two\_filter={field:"parentId", operator:"eq", value: parseInt(secondDropDownList.dataItem().regionId)};    thirdDropDownList.dataSource.filter(two\_filter); |

最后定义 change 函数，当下拉框的值改变时调用。在此重新定义过滤器，使每次改变值的时候下级联动都会随之改变。

|  |
| --- |
| //省  change:function(){  //修改值后更新下拉列表2和下拉列表3数据  var one\_filter={field:"parentId", operator:"eq", value: parseInt(firstDropDownLis  t.dataItem().regionId)};  secondDropDownList.dataSource.filter(one\_filter);  secondDropDownList.select(-1);  thirdDropDownList.select(-1);  }  //市  change:function(){  var two\_filter={field:"parentId", operator:"eq", value: parseInt(secondDropDown  List.dataItem().regionId)};  thirdDropDownList.dataSource.filter(two\_filter);  thirdDropDownList.select(-1);  } |

如图所示: 

## 前端开发案例

### 单表操作

#### 整体界面效果展示



#### 具体前端实现

1. 头html文件准备及引入

在实际项目开发中，往往需要在html中引入较多外部js、css、jquery等文件，而且不同的html页面需要引入的大多都一样。所以一般会把这部分头代码单独放在一个html中。在其他地方需要用的话直接导入即可，方便又简洁。如本例中的header.html。header.html的具体代码如下：

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <title>${SYS\_TITLE!'Hand Application Platform'}</title>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=utf-8"*/>  <meta http-equiv=*"X-UA-Compatible"* content=*"IE=edge,chrome=1"*>  <link href=*"${base.contextPath}/resources/upload/favicon.png"* rel=*"shortcut icon"* />  <!--[if IE 8]>  <script src="${base.contextPath}/lib/polyfill/respond.min.js"></script>  <script src="${base.contextPath}/lib/polyfill/es5-shim.min.js"></script>  <![endif]-->  <script src=*"${base.contextPath}/lib/kendoui/js/jquery.min.js"*></script>  <script src=*"${base.contextPath}/lib/kendoui/js/kendo.all.min.js"*></script>  <link href=*"${base.contextPath}/lib/bootstrap-3.3.7/css/bootstrap.min.css"* rel=*"stylesheet"* type=*"text/css"*/>  <link href=*"${base.contextPath}/lib/font-awesome-4.6.3/css/font-awesome.min.css"* rel=*"stylesheet"* type=*"text/css"*/>  <link href=*"${base.contextPath}/lib/kendoui/styles/kendo.common-bootstrap.min.css"* rel=*"stylesheet"* type=*"text/css"*/>  <link href=*"${base.contextPath}/lib/kendoui/styles/kendo.bootstrap.min.css"* rel=*"stylesheet"* type=*"text/css"*/>  <link href=*"${base.contextPath}/lib/kendoui/styles/kendo.hap.css"* rel=*"stylesheet"* type=*"text/css"*/>  <script src=*"${base.contextPath}/lib/kendoui/js/cultures/kendo.culture.${base.locale.toString()?replace('\_','-')}.js"*></script>  <script src=*"${base.contextPath}/lib/kendoui/js/messages/kendo.messages.${base.locale.toString()?replace('\_','-')}.js"*></script>  <script src=*"${base.contextPath}/lib/kendoui/js/kendo.hap.js"*></script>  <script src=*"${base.contextPath}/common/prompts"*></script>  <script src=*"${base.contextPath}/resources/js/pdfobject.js"*></script>  <link href=*"${base.contextPath}//resources/css/frm/util.css"* rel=*"stylesheet"* type=*"text/css"*/>  <script>  \_basePath = '${base.contextPath}',\_locale='${base.locale}';  kendo.culture("${base.locale.toString()?replace('\_','-')}");  </script>  <script src=*"${base.contextPath}//resources/js/noChinese.js"*></script>  </head>  <#macro lov lovid>${lovProvider.getLov(base.contextPath,base.locale,lovid)}</#macro> |

1. 前端完整代码展示以及各个部分代码用途说明

|  |
| --- |
| **<!—引入头文件-->**  <#include "../include/header.html">  <#include "../include/hss.html">  <body>  <script>  **//准备viewModel**  viewModel = kendo.observable({  //viewModel一般在body的最前面声明，因为页面中按钮、字段等都需要与其绑定  model : {},  createFunction: **function**(){//新建操作函数  $('#process\_node\_grid').data('kendoGrid').addRow();  },  saveFunction: **function**(){//更新或者新建保存函数  $('#process\_node\_grid').data('kendoGrid').saveChanges();  },  queryResource : **function** (e) {//查询函数  $('#process\_node\_grid').data('kendoGrid').dataSource.page(1);  },  resetForm : **function** (e) {//查询条件重置函数  **var** formData = viewModel.model.toJSON();  **for** (**var** k **in** formData) {  viewModel.model.set(k, **null**);  }  }  });  </script>  <div id=*"page-content"*>  **<!—操作按钮标签声明：即页面中的新建、保存、删除-->**  <div class=*"pull-left"* id=*"toolbar-btn"* style="padding-bottom:*10px*;">  <span class=*"btn btn-primary k-grid-add"* style="float:*left*;margin-right:*5px*;" data-bind=*"click:createFunction"*><@spring.message "hap.new"/></span>  <span class=*"btn btn-success k-grid-save-changes"* data-bind=*"click:saveFunction"* style="float:*left*;margin-right:*5px*;"><@spring.message "hap.save"/></span>  <span onclick="deleteData()" class=*"btn btn-danger"* style="float:*left*;margin-right:*5px*;"><@spring.message "hap.delete"/></span>  </div>  <script>  /\*将按钮绑定到viewModel，也可以先不绑定，到最后直接把跟标签与viewModel绑定，比如本界面中id为*page-content的div\*/*  kendo.bind($('#toolbar-btn'), viewModel);  </script>  <**!—查询条件form-->**  <div class=*"pull-right"* id=*"process\_node\_form"* style="padding-bottom:*10px*;">  <input type=*"text"* style="float:*left*;width:*200px*;margin-right:*5px*;" placeholder=*'<@spring.message "hss.setup.processnode.processnodename"/>'* data-bind=*"value:model.processNodeName"* class=*"k-textbox"*>  <input type=*"text"* style="float:*left*;width:*150px*;margin-right:*5px*;" placeholder=*'<@spring.message "hap.description"/>'* data-bind=*"value:model.description"* class=*"k-textbox"*>  <span class=*"btn btn-primary"* style="float:*left*;width:*70px*;margin-right:*5px*;" data-bind=*"click:queryResource"* type=*"submit"*><@spring.message "hap.query"/></span>  <span class=*"btn btn-default"* style="float:*left*;width:*70px*" data-bind=*"click:resetForm"* type=*"button"*><@spring.message "hap.reset"/></span>  <div style="clear:*both*"></div>  </div>  <script>kendo.bind($('#process\_node\_form'), viewModel);</script>  dto字段绑定  <div style="clear:*both*">  **<!—用一个空的div来承载grid -->**  <div id=*"process\_node\_grid"*></div>  </div>  </div>  点击事件函数绑定  <script>  //数据验证绑定  $("#page-content").kendoValidator();  //查询条件框回车查询  $('#process\_node\_form input').keydown(**function** (e) {  **if** (e.keyCode == 13) {  e.target.blur();  viewModel.queryResource(e);  }  });  //grid数据源配置  **var** dataSource = **new** kendo.data.DataSource({  transport : {  read : {//查询配置  url : "${base.contextPath}/setup/processNode/query",  type : "POST",  dataType: "json"  },  create : {//新建配置  url : "${base.contextPath}/setup/processNode/submit",  contentType: "application/json",  type : "POST"  },  update : {//修改配置  url : "${base.contextPath}/setup/processNode/submit",  contentType: "application/json",  type : "POST"  },  destroy : {//删除配置  url : "${base.contextPath}/setup/processNode/delete",  contentType: "application/json",  type : "POST"  },  parameterMap: **function** (options, type) {  **if** (type !== "read" && options.models) {  //去除前后空格  **for**(**var** i=0;i<options.models.length;i++){  options.models[i].processNodeName=options.models[i].processNodeName.trim();  options.models[i].description=options.models[i].description.trim();  }  //提交参数处理  **var** datas = Hap.prepareSubmitParameter(options, type)  **return** kendo.stringify(datas);  } **else** **if** (type === "read") {  //查询参数处理  **return** Hap.prepareQueryParameter(viewModel.model.toJSON(), options)  }  }  },  requestEnd: **function**(e) {//请求结束回调处理  **if** (e.response.success && e.type != "read") {  kendo.ui.showInfoDialog({  //描述维护的信息代码，显示的是根据当前环境下的消息  message:'<@spring.message "hap.tip.success"/>'  })  }  **if**(e.type != "read"){  dataSource.page(dataSource.\_page);  }  },  requestEnd : **function**(e) {  **if** (e.response.success && e.type != "read") {  kendo.ui.showInfoDialog({  message:'<@spring.message "hap.tip.success"/>'  })  }  },  batch : **true**,//支持批量操作  serverPaging: **true**,//支持服务端分页  pageSize : 10,  schema : {  data : 'rows',  total : 'total',  model : {  id : "processNodeId",  fields: {  processNodeName : {validation: {required: **true**}},  description : {validation: {required: **true**}},  enabledFlag: {defaultValue: 'Y',type: 'boolean',checkedValue:'Y',uncheckedValue:'N'},  }  }  }  });  **var** grid = $("#process\_node\_grid").kendoGrid({  dataSource : dataSource,  height : '100%',  resizable : **true**,  columnMenu: **true**, //可以藏列  reorderable: **true**,//可调整列顺序  sortable : **true**,  scrollable : 'multiple,rowbox',  navigatable: **false**,  selectable :'multiple,rowbox',  pageable : {  必输校验  pageSizes : [5, 10, 20, 50],  refresh : **true**,  buttonCount: 5  },  editable : "inline",//按行编辑  columns : [{  field : "processNodeName",  title : '<@spring.message "hss.setup.processnode.processnodename"/>',  editor: **function**(container, options){  **var** editor = $('<input type="text" style="padding:0;height:25px;" name="processNodeName"  class="k-input k-text-box" required validationMessage="<@spring.message "hss.notempty"/>">');  editor.css('width','100%');  container.append(editor);  },  校验失败提示消息  width : 200  },{  field : "description",  title : '<@spring.message "hap.description"/>',  width : 200,  editor: **function** (container, options) {  $('<input name="' + options.field + '"/>')  .appendTo(container)  .kendoTLEdit({  idField: 'processNodeId',  field: 'description',  dto: "hss.setup.dto.ProcessNode",  model: options.model  });  }  },{  field :"enabledFlag",  title : '<@spring.message "hss.enabled\_flag"/>',  width : 100,  attributes : {style: "text-align:center"}  },{  command: [{name:"edit"},{name:"destroy"}],  title: " ",  attributes : {style: "text-align:center"},  width: "100px"  }]  }).data("kendoGrid");  **function** deleteData() {//自定义批量删除函数  **var** checked = grid.selectedDataItems();//获取选中的行  **if**(grid.selectedDataItems().length){  kendo.ui.showConfirmDialog({  title:$l('hap.tip.info'),  message: $l('hap.tip.delete\_confirm')  }).done(**function** (event) {  **if** (event.button == 'OK') {  $.ajax({  type : "POST",  url : "${base.contextPath}/setup/processNode/delete",  data : kendo.stringify(checked),//数据转json  dataType:"json",  contentType:"application/json",  success: **function**(e) {  **if**(e.message){  kendo.ui.showInfoDialog({  message:e.message  })  }**else**{  $.each(checked,**function**(i,v){  grid.dataSource.remove(v)  });  kendo.ui.showInfoDialog({  message:'<@spring.message "hap.tip.success"/>'  })  }  }  });  }  })  }**else**{  kendo.ui.showInfoDialog({  message:$l('hap.tip.selectrows')  })  }  };  $("#process\_node\_grid thead>tr th").css("text-align","center");  Hap.autoResizeGrid("#process\_node\_grid");//允许grid自适应浏览器  </script>  </body>  </html> |

#### 后台代码实现

1. Controller层

|  |
| --- |
| @Controller  **public** **class** ProcessNodeController **extends** BaseController {  @Autowired  **private** IProcessNodeService processNodeService;  @Autowired  **private** MessageSource messageSource;  /\*\*  \* 处理节点查询.  \* **@param** processnode 分配对象  \* **@param** page 起始页  \* **@param** pagesize 分页大小  \* **@param** request HttpServletRequest  \* **@return** ResponseData  \*/  @RequestMapping(value = "/setup/processNode/query ",method = RequestMethod.***POST***)  @ResponseBody  **public** ResponseData getProcessNodeInfo(ProcessNode node, @RequestParam(defaultValue = ***DEFAULT\_PAGE***) **int** page,  @RequestParam(defaultValue = ***DEFAULT\_PAGE\_SIZE***) **int** pagesize, HttpServletRequest request) {  IRequest requestContext = createRequestContext(request);  List<ProcessNode> list=processNodeService.selectProcesNode(requestContext, node, page, pagesize);  **return** **new** ResponseData(list);  }    /\*\*  \* 保存处理节点更新信息.包括：新增、更新  \* **@param** nodeList processNode集合  \* **@param** result BindingResult  \* **@param** request HttpServletRequest  \* **@return** ResponseData ResponseData  \* **@throws** BaseException BaseException  \*/  @RequestMapping(value = "setup/processNode/submit", method = RequestMethod.***POST***)  @ResponseBody  **public** ResponseData submitProcessNodeInfo(@RequestBody List<ProcessNode> nodeList, BindingResult result, HttpServletRequest request)  **throws** BaseException {  //非空性验证，方法继承自BaseController  getValidator().validate(nodeList, result);  **if** (result.hasErrors()) {  ResponseData rd = **new** ResponseData(**false**);  rd.setMessage(getErrorMessage(result, request));  **return** rd;  }  //唯一性验证  Locale locale = RequestContextUtils.*getLocale*(request);  String errorTip=processNodeService.uniqueCheck(nodeList, locale);  **if**(!errorTip.isEmpty()){  ResponseData rd = **new** ResponseData(**false**);  rd.setMessage(errorTip);  **return** rd;  }  //验证通过，操作数据库  IRequest requestContext = createRequestContext(request);  List<ProcessNode> resultList;  resultList = processNodeService.batchUpdate(requestContext, nodeList);  **return** **new** ResponseData(resultList);  }    /\*\*  \* 删除处理节点  获取当前语言环境下的错误提示消息  \* **@param** node  \* **@param** request  \* **@return**  \*/  @RequestMapping(value = "setup/processNode/delete", method = RequestMethod.***POST***)  @ResponseBody  **public** ResponseData deleteProcessNode(@RequestBody List<ProcessNode> list,  HttpServletRequest request){  Locale locale = RequestContextUtils.*getLocale*(request);  IRequest requestContext = createRequestContext(request);  String result=processNodeService.deleteProcessNode(requestContext,list);  **if**(!result.isEmpty()){  ResponseData rd=**new** ResponseData(**false**);  rd.setMessage(messageSource.getMessage(result, **null**, locale));  **return** rd;  }  **return** **new** ResponseData();  }  } |

1. Service接口层

|  |
| --- |
| **public** **interface** IProcessNodeService **extends** IBaseService<ProcessNode>, ProxySelf<IProcessNodeService>{  /\*\*  \* 处理节点唯一性检验  \* **@param** node  \* **@return** 错误提示代码，检验通过则返回空  \*/  **public** String uniqueCheck(List<ProcessNode> nodeList,Locale locale);  /\*\*  \* 删除处理节点  \* **@param** requestContext  \* **@param** nodeList  \* **@return** List<ProcessNode>  \* **@throws** ValidationTableException  \*/  **public** String deleteProcessNode(IRequest requestContext,@StdWho List<ProcessNode> nodeList);  /\*\*  \* 按条件查询处理节点（名称、描述，支持模糊查询）  \* **@param** requestContext  \* **@param** page  \* **@param** pagesize  \* **@param** node  \* **@return** List<ProcessNode>  \*/  **public** List<ProcessNode> selectProcesNode(IRequest requestContext,ProcessNode node, **int** page, **int** pagesize);  } |

1. Service实现层

|  |
| --- |
| @Service  **public** **class** ProcessNodeServiceImpl **extends** BaseServiceImpl<ProcessNode> **implements** IProcessNodeService {  @Autowired  **private** ProcessNodeMapper processNodeMapper;  @Autowired  **private** MessageSource messageSource;  @Autowired  WHO字段：lastUpdateLogin、lastUpdateDate、lastUpdatedBy、creationDate、createdBy处理  **private** TaskNodeMapper taskNodeMapper;  /\*\*  \* **@description** 处理节点修改以及新增批处理  \* **@param** List<ProcessNode>  \* **@return** nodeList  \*/  @Override  @Transactional(propagation = Propagation.***REQUIRED***,rollbackFor=ValidationTableException.**class**)  **public** List<ProcessNode> batchUpdate(IRequest requestContext,@StdWho List<ProcessNode> nodeList) {  **for** (ProcessNode node : nodeList) {  **if** (node.get\_\_status() != **null**) {  **switch** (node.get\_\_status()) {  **case** DTOStatus.***ADD***:  processNodeMapper.insertSelective(node);  **break**;  **case** DTOStatus.***UPDATE***:  processNodeMapper.updateByPrimaryKeySelective(node);  **break**;  **default**:  **break**;  }  }  }  **return** nodeList;  }  /\*\*  \* 删除处理节点  \* **@param** requestContext  \* **@param** nodeList  异常提示信息描述维护代码  \* **@return** List<ProcessNode>  \* **@throws** ValidationTableException  \*/  @Override  @Transactional(propagation = Propagation.***REQUIRED***,rollbackFor=ValidationTableException.**class**)//事务控制：乐观锁  **public** String deleteProcessNode(IRequest requestContext,@StdWho List<ProcessNode> list){  TaskNode taskNode=**new** TaskNode();  **for**(ProcessNode processNode:list){  taskNode.setProcessNodeId(processNode.getProcessNodeId());  **int** resut=taskNodeMapper.processNodeIsUsedCheck(taskNode);  **if**(resut==0){  processNodeMapper.delete(processNode);  }**else**{  **return** "hss.setup.processnode.node\_cancel\_error";  }  }  **return** **null**;  }  /\*\*  \* **@description** 处理节点名称唯一性检验  \* **@param** node  \* **@return** 错误提示代码，没有则为null  \*/  **public** String checkUniqueForUpdate(ProcessNode node){  **int** i=processNodeMapper.checkDescriptionUnique(node);  **if**(i==1){  **return** "hss.setup.process\_node.description\_error";  }**else**{  **return** "";  }  }  /\*\*  \* **@description** 处理节点描述唯一性检验  \* **@param** node  \* **@return** 错误提示代码，正确则返回null  \*/  **public** String checkUniqueForInsert(ProcessNode node){  **int** i=processNodeMapper.checkNameUnique(node);  **int** j=processNodeMapper.checkDescriptionUnique(node);  **if**(i>0){  **return** "hss.setup.process\_node.node\_name\_error";//"该节点名称已存在";  }**else** **if**(j>0){  **return** "hss.setup.process\_node.description\_error";  }**else**{  **return** "";  }  }  /\*\*  \* **@description** 处理节点批处理描述唯一性检验(在数据库检验之前)  \* **@param** node  \* **@return** 错误提示代码，没有则为空  \*/  **public** String checkUniqueBeforeDB(List<ProcessNode> nodeList){  String errorTip="";  **for**(**int** i=0;i<nodeList.size();i++){  **for**(**int** j=i+1;j<nodeList.size();j++){  **if**(nodeList.get(i).getProcessNodeName().equals(nodeList.get(j).getProcessNodeName())){  errorTip="hss.setup.process\_node.node\_name\_error";  **break**;  }  **if**(nodeList.get(i).getDescription().equals(nodeList.get(j).getDescription())){  errorTip="hss.setup.process\_node.description\_error";  **break**;  }  }  }  **return** errorTip;  }  /\*\*  \* **@description** 处理节点批处理唯一性检验  \* **@param** node  \* **@return** 错误提示代码，没有则为空  \*/  @Override  **public** String uniqueCheck(List<ProcessNode> nodeList,Locale locale){  String tip="";  //唯一性校验（操作数据库之前）  String error=checkUniqueBeforeDB(nodeList);  **if**(!error.isEmpty()){  tip=messageSource.getMessage(error,**null**, locale);  **return** tip;  }  //唯一性校验（进入数据库检验）  **for**(ProcessNode node: nodeList){  **if**(node.get\_\_status().equals(DTOStatus.***UPDATE***)){  error=checkUniqueForUpdate(node);  **if**(!error.isEmpty()){  tip=messageSource.getMessage(error,**null**, locale);  **return** tip;  }  }  **if**(node.get\_\_status().equals(DTOStatus.***ADD***)){  error=checkUniqueForInsert(node);  **if**(!error.isEmpty()){  tip=messageSource.getMessage(error,**null**, locale);  **return** tip;  }  }  }  **return** tip;  }  /\*\*  \* **@description** 按条件查询处理节点（名称，描述，支持模糊查询）  \* **@param** requestContext  \* **@param** page  \* **@param** pagesize  \* **@param** node  \* **@return**  \*/  **public** List<ProcessNode> selectProcesNode(IRequest requestContext, ProcessNode node, **int** page, **int** pagesize){  PageHelper.*startPage*(page, pagesize);//分页  **return** processNodeMapper.selectProcessNodeBY(node);    }  } |

1. Mapper接口层

|  |
| --- |
| **public** **interface** ProcessNodeMapper **extends** Mapper<ProcessNode> {  /\*\*  提示：对单表基本的增、删、改、查操作，直接调用框架封装的方法即可，没必要自己再写。  \* **@description** 处理节点名称唯一性检验  \* **@param** node  \* **@return** 返回记录行数  \*/  **public** **int** checkNameUnique(ProcessNode node);  /\*\*  \* **@description** 处理节点更新描述唯一性检验  \* **@param** node  \* **@return** 返回记录行数  \*/  **public** **int** checkDescriptionUnique(ProcessNode node);  /\*\*  \* **@description** 按条件查询处理节点  \* **@param** ProcessNode node  \* **@return** List<ProcessNode>  \*/  **public** List<ProcessNode> selectProcessNodeBY(ProcessNode node); |

1. Mapper的XML配置

|  |
| --- |
| <?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=*"hss.setup.mapper.ProcessNodeMapper"* >  <resultMap type=*"hss.setup.dto.ProcessNode"* id=*"BaseResultMap"* extends=*"com.hand.hap.mapper.StdMapper.STD"*>  <result column=*"PROCESS\_NODE\_ID"* property=*"processNodeId"* jdbcType=*"DECIMAL"*/>  <result column=*"PROCESS\_NODE\_NAME"* property=*"processNodeName"* jdbcType=*"VARCHAR"*/>  <result column=*"DESCRIPTION"* property=*"description"* jdbcType=*"VARCHAR"*/>  <result column=*"ENABLED\_FLAG"* property=*"enabledFlag"* jdbcType=*"VARCHAR"*/>  <result column=*"DESCRIPTION"* property=*"description"* jdbcType=*"VARCHAR"*/>  </resultMap>  <!-- 处理节点名称唯一性校验 -->  <select id=*"checkNameUnique"* resultType=*"int"* parameterType=*"hss.setup.dto.ProcessNode"*>  SELECT COUNT(PROCESS\_NODE\_ID)  FROM HSS\_SETUP\_PROCESS\_NODE\_B  WHERE PROCESS\_NODE\_NAME=#{processNodeName,jdbcType=VARCHAR}  <if test=*"processNodeId != null"*>  AND PROCESS\_NODE\_ID &lt;&gt;#{processNodeId,jdbcType=DECIMAL}  </if>  </select>  <!-- 处理节点描述（状态）唯一性校验 -->  <select id=*"checkDescriptionUnique"* resultType=*"int"* parameterType=*"hss.setup.dto.ProcessNode"*>  SELECT COUNT(PROCESS\_NODE\_ID)  FROM HSS\_SETUP\_PROCESS\_NODE\_B  WHERE DESCRIPTION=#{description,jdbcType=VARCHAR}  <if test=*"processNodeId != null"*>  AND PROCESS\_NODE\_ID &lt;&gt;#{processNodeId,jdbcType=DECIMAL}  </if>  </select>  <!-- 处理节点条件查询 -->  <select id=*"selectProcessNodeBY"* parameterType=*"hss.setup.dto.ProcessNode"* resultMap=*"BaseResultMap"*>  SELECT HSPN.PROCESS\_NODE\_ID,  HSPN.PROCESS\_NODE\_NAME,  HSPN.ENABLED\_FLAG,  HSPN\_TL.DESCRIPTION  FROM HSS\_SETUP\_PROCESS\_NODE\_B HSPN,  HSS\_SETUP\_PROCESS\_NODE\_TL HSPN\_TL  WHERE HSPN.PROCESS\_NODE\_ID = HSPN\_TL.PROCESS\_NODE\_ID  AND HSPN\_TL.LANG = #{request.locale,jdbcType=VARCHAR}  <if test=*"processNodeName!= null"*>  AND HSPN.PROCESS\_NODE\_NAME LIKE #{processNodeName,jdbcType=VARCHAR}  </if>  <if test=*"description!=null"*>  AND HSPN\_TL.DESCRIPTION LIKE #{description,jdbcType=VARCHAR}  </if>  <if test=*"enabledFlag != null"*>  AND HSPN.ENABLED\_FLAG = #{enabledFlag,jdbcType=VARCHAR}  </if>  </select>  </mapper> |

1. DTO层

|  |
| --- |
| @Table(name = "HSS\_SETUP\_PROCESS\_NODE\_B")  @MultiLanguage  **public** **class** ProcessNode **extends** BaseDTO{  @Id  @GeneratedValue(generator = ***GENERATOR\_TYPE***)  @Column  **private** Long processNodeId;  说明：如果是头行结构，只需要将行表的List作为一个属性声明在头dto中，并加上  @Trancient注解和@Children注解即可。例  @Childre  @Transient  private List<行dto> list;  @Column  @NotEmpty  **private** String processNodeName;  @Column  @NotEmpty  @MultiLanguageField  **private** String description;  @Column  @NotEmpty  **private** String enabledFlag;  **public** ProcessNode() {  **super**();  }  } |

### 头行结构

#### 头表单提交

如果只是提交头信息，则可以直接调用LCP封装的submitForm方法即可。

例：

|  |
| --- |
| submitHeaderForm =**function**(e){  viewModel.model.set("operationItem","create");  Hap.submitForm({  url: '${base.contextPath}/elp/expressManage/headerSubmit',  formModel: viewModel.model,  success: **function** (json) {  **var** data=json.rows[0]||{};  **for**(**var** k **in** data ){  viewModel.model.set(k, data[k]);  }  kendo.ui.showInfoDialog({  message:'<@spring.message "hap.tip.success"/>'  })  }  });  } |

#### 头行同时提交

在kendo UI中，不管是头数据还是行数据都绑定在viewModel的model中。所以在提交行数据（即grid数据）时，把model里的数据传到后台即可。其他操作跟非头行结构的一样。

## 常见问题及解决办法

## 未结与已结问

### 未结问题

| 序号 | 问题 | 解决方案 | 负责人 | 目标日期 | 实际日期 |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

### 已结问题

| 序号 | 问题 | 解决方案 | 负责人 | 目标日期 | 实际日期 |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |