使用antelope

目录

[1. 前言和概述 1](#_Toc342421824)

[1) 前言 1](#_Toc342421825)

[2) 框架概述 2](#_Toc342421826)

[2. 全界面组件 3](#_Toc342421827)

[1) 概述 3](#_Toc342421828)

[2) 快速使用 3](#_Toc342421829)

[3. 从零开始创建我们的子项目 7](#_Toc342421830)

[1) 设置项目首页及登录页 7](#_Toc342421831)

[2) 项目相关的命名规范 9](#_Toc342421832)

[3) 使用ant更新平台代码 10](#_Toc342421833)

[4) 使用Visual Paradigm软件绘制UML 11](#_Toc342421834)

[5) 创建sql脚本更新文件 13](#_Toc342421835)

[6) 添加模块导航 14](#_Toc342421836)

[7) 添加一个基本数据请求响应 15](#_Toc342421837)

[4. 数据库数据的查询操作相关接口 18](#_Toc342421838)

[5. 表单控件以及表单验证 19](#_Toc342421839)

[1) 框架内置的class类型控件 20](#_Toc342421840)

[2) 表单验证 23](#_Toc342421841)

[6. 计划任务 24](#_Toc342421842)

[1) Spring注解方式 24](#_Toc342421843)

[2) 接口方式 24](#_Toc342421844)

[7. 附录 框架主要接口API 27](#_Toc342421845)

[3) JPABaseDao 27](#_Toc342421846)

[4) DBUtil 30](#_Toc342421847)

# 前言和概述

## 前言

我是李宁。

在具体写框架使用之前说两句题外话，相信在软件开发的这条道路上每个人都有每个人的思考和想法，那其实在具体工作当中，除了那些大脑很发达，能去到微软谷歌一般的开发人员之外，其实更多的是我们这些每天加班加点，却又默默无闻的普通开发人员。

而我一直相信，优秀的软件，是由一群优秀的开发人员开发出来的，只有开发人员自己真正做到了优秀，才能做出优秀的软件。

那么我们面临的困难很显然，就在于我们不能安于现状，让普通的自己变成一个优秀的开发人员。

在传统体制条件下，命令执行模式让更多的人经受到了压力，却没有释放出自己的潜力，而我一贯认为，在轻松快乐的环境下更能激发员工的灵感，让自己成为一个真正的技术人员，而不是所谓的IT民工，是我，也是我所希望的未来团队所有人能够感受到的自我认知。

我搭建的antelope平台所有架构设计，都源于几个最基本的观念：

首先，必须要让大家不仅能够开发出软件，而且能够从框架本身学到东西。

为了实现这个目标，我抛弃了当前流行的maven管理模式，而是直接采用源码覆盖的方式进行框架集成，这样所有参与人员都可以直接打开框架源码查看和尝试修改，为了保证代码质量，框架级代码修改均为临时修改，是无效的，当再次将框架更新完毕之后会将之前的修改覆盖掉，但是不影响大家对源码的深入研究。

其次，为了让大家能够更高效率地做应用开发，我在框架当中开发了相当多的常用组件。

有一些有界面，有一些没有，我的目标是在3到5年之内将我们的软件开发周期缩短到最小，我们总体的软件质量提升到一个新的阶段，让我们将更多的时间投入到对技术的探索当中，而不是浪费在平时的重复性劳动当中。

第三，保持永久的版本兼容性和可扩展性，不仅仅要向平台持续注入新的活力，而且要让子项目在平台更新过程当中天然受益。

平台代码能否保证在未来能继续扩展，却能够在子项目更新平台代码时不至于影响子项目现有功能是我评价平台代码合格性的很重要的一个指标。没有人能完全达到这个目标，即使是微软也出现过一些兼容性问题，但是它发生的频率越低，修复缺陷所带来的成本就会越低，它是一个正确的方向。

有了这些基本的架构指导思想，我相信，未来我们团队的优秀人才能够不断为平台添砖加瓦，注入新的活力，无论是在功能上，还是在性能上，可靠性上或者是在兼容性等各个方面都能做出贡献。

## 框架概述

antelope框架利用了当前J2EE平台最流行，最稳定的基础框架技术，其中包括Hibernate的JPA数据访问接口，springMVC框架（包括AOP切面切入事务等）；前台使用了当前互联网使用率最高的jQuery框架技术，包括jQuery-ui界面框架的使用。除此之外，数据库升级使用dbdeployer技术，流程引擎使用了当前流行的轻量级activiti引擎，图表我们使用了基于flash的基础flashchart和酷炫的fusionchart。在框架之外，我们利用ant进行平台代码升级，使用CruiseControl进行代码的持续集成。

在上述技术的充分利用之下，我们进行了一系列的代码封装，包括最常用的ajax前后台分页，表头排序技术，基于flash的单文件、多文件上传组件，甚至部分基于jni和c代码生成，为了获取windows底层数据的部分接口，还有非常全面丰富的数据库访问接口,sql代码执行接口，所有这些封装的接口均可以在项目中跟踪其源码，并且主要的接口都附有注释，可以进行查阅。

除了基础接口的封装，在平台级别我们还具备一些平台的基础功能，包括角色权限管理、组织机构人员管理、数据迁移及计划任务管理等。

antelope平台除了拥有上述看得见的功能之外，基于强大的spring 注解模式，平台拥有了非常灵活的基础功能低耦合式架构，基于后台事件模型和前台的基于扩展点的插件模型，允许基于平台开发的子项目在不修改任何平台代码的前提下做任意扩展，架构的灵活性给了平台一个健康发展的前提。

综上，我们可以快速基于平台开发子项目，不断提升我们的技术储备，提升公司J2EE团队的生产力。

# 全界面组件

## 概述

antelope快速开发平台将最常用的通用功能抽象出来组件化，来提高系统的开发效率，并提高系统整体的修改灵活性，全界面组件为平台最高级的一类组件，为快速开发包括基本增删改查以及流程化功能提供了最基本的代码结构。

全界面组件主类均保存在antelope.interfaces.components包中。

组件所引用的jsp文件均保存在系统webapp/components目录中。

全界面组件包括如下两类共五个基本组件(文档中，后面主类描述默认使用主类所在包，不再书写全包)：

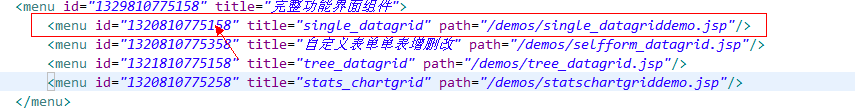
|  |  |  |  |
| --- | --- | --- | --- |
| **类别** | **主类** | **主jsp** | **完成功能** |
| js接口全界面组件 | ListSelect | listselect.jsp | 完成列表选择功能，单选、多选、条件查询 |
| jsp接口全界面组件 | SingleDatagrid | single\_datagrid.jsp | 类单表增删改查。 |
| TreeDatagrid | tree\_datagrid.jsp | 类左树右列表关联增删改查 |
| WorkflowDatagrids | workflow\_datagrids.jsp | 工作流创建，查看、完成用户节点任务。 |
| StatsChartgrid | stats\_chartgrid.jsp | 左表右图，含有查询导出的统计图表功能。 |

## 快速使用

以SingleDatagrid为例使用时所涉及到如下几个文件：

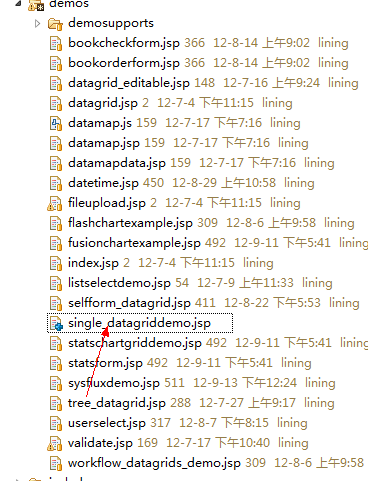
|  |  |
| --- | --- |
| 文件 | 作用 |
| SingleDatagrid.java | 继承此类，完成全界面组件后台相关操作 |
| i18n/messages\_components\_zh\_CN.properties | 使用此类完成具体功能中字符串的国际化工作 |
| single\_datagrid.jsp | 使用此jsp完成全界面组件前台代码的引用工作 |
| SingleDatagridOptions等其他类文件 | 与SingleDatagrid一起完成模块独有的可选项功能 |

创建一个新的模块按照以下步骤进行：

1. 在module-cfg\_zh\_CN.xml中添加中文目录项如下  
   

其中注意箭头所指向的id值，要求在module-cfg\_zh\_CN.xml文档中全局惟一。

1. 按照所添加的目录对应地新建出模块jsp文件



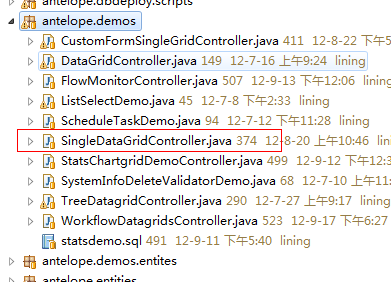
1. jsp文件当中拷贝入如下内容

<%@ page language=*"java"* pageEncoding=*"utf-8"*%>

<jsp:include page=*"/components/single\_datagrid.jsp"*>

<jsp:param name=*"component"* value=*"singledatagriddemo"*/>

</jsp:include>

1. 其中加红部分为功能模块的前后台对应关系的字符串，需要书写一个对于模块有一定含义，并且全局唯一的字符串。
2. 创建类继承自SingleDatagrid如下：  
     
   注意所创建的Java类需要使用Controller结尾，以自动打开类中方法的事务。

Demo中全界面组件代码如下：

/\*\*

\* 单列表增删改全功能

\* **@author** lining

\* **@since** 2012-7-14

\*/

@Controller("singledatagriddemo")

@RequestMapping("/demos/single\_datagrid/SingleDataGridController")

**public** **class** SingleDataGridController **extends** SingleDatagrid {

**public** **void** getSingleGridList(String name, HttpServletRequest req, HttpServletResponse res) **throws** SQLException, Exception {

SqlWhere sqlwhere = tidySqlWhere(**new** String[]{"name like ?"}, **new** Object[]{"%" + *decodeAndTrim*(name) + "%"});

PageItem pageItem = DBUtil.*queryJSON*("select \* from DEMO\_SINGLE\_DATAGRID where 1=1 " + sqlwhere.wherePart, sqlwhere.outParams, getPageParams(req));

print(pageItem, res);

}

**public** **void** deleteOneLine(String sid, HttpServletRequest req, HttpServletResponse res) **throws** InstantiationException, IllegalAccessException {

dao.deleteBy(sid, SingleDataGridItem.**class**);

}

**public** **void** addOrUpdateOne(String sid, HttpServletRequest req, HttpServletResponse res) **throws** Exception {

SingleDataGridItem item = dao.getBy(sid, SingleDataGridItem.**class**);

**if** (item == **null**)

item = newInstanceWithCreateInfo(SingleDataGridItem.**class**, req);

*wrapToEntity*(req, item);

dao.insertOrUpdate(item);

}

@Override

**public** SingleDatagridOptions getOptions(HttpServletRequest req) {

SingleDatagridOptions opts = **new** SingleDatagridOptions(**this**);

opts.showCreateBtn = **true**;

opts.queryfields = **new** String[]{"name"};

opts.formfields = **new** FormField[]{**new** FormField("name", "名字", "required='true' maxlength2='10'")};

opts.columns.put("name", **new** GridColumn("名字", "50%"));

**return** opts;

}

}

其中getOptions部分用来为模块定制列表列对应管理，以及界面组件可选项。

getSingleGridList用来在列表初始化时获取列表数据。

deleteOneLine用来当点击列表中的删除按钮时调用删除后台数据。

addOrUpdateOne用来在用户点击创建按钮或者点击单行中的编辑按钮时新建或修改记录时调用的后台方法。

值得注意的是类上方的注解@Controller("singledatagriddemo")

其中singledatagriddemo为创建jsp时所添加进入的component参数，使用它来完成前台jsp和后台java类的调用对接过程。

另外，为了完成对后台程序的权限控制，在url方面需要遵循一个规范，如本例中类上方的注解@RequestMapping("/demos/single\_datagrid/SingleDataGridController")

其中/demos/single\_datagrid为对应创建的jsp文件的全路径，将.jsp结尾去掉。然后添加上对应Controller类名称。

1. 下面完成模块的国际化工作，即使模块不需要其他语言的支持也需要将基本的中文的完成

打开messages\_components\_zh\_CN.properties组件国际化属性文件之后在上方可以看到如下配置信息

#组件演示

####single\_datagrid

single\_datagrid.singledatagriddemo.add=添加演示

single\_datagrid.singledatagriddemo.modify=修改演示

single\_datagrid.singledatagriddemo.view=查看演示

1. 将此三行拷贝下来，再打开messages\_project\_zh\_CN.properties文件，将其粘贴到一个合适的位置，然后将字符串中两个点中间的字符串更换为对应模块的关键字。
2. 将对应的中文描述修改为模块对应的中文描述。
3. 至此，完成SingleDatagrid全界面组件的基本开发使用工作。

## 详细功能说明

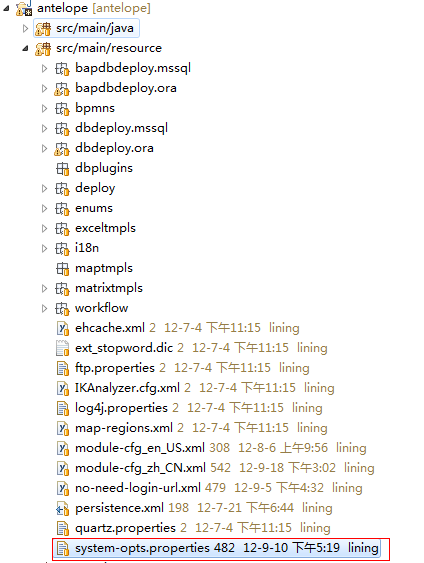
### 电风扇

# 从零开始创建我们的子项目

## 设置项目首页及登录页

在开发系统时,我们会遇到各式各样的系统,那么系统中首先可能会有所不同的就是系统的登录页面和首页面。例如有的登录页面仅仅需要一个允许输入用户名和密码和进行登录的页面，有些系统的登录页面可能需要一个能够展示在用户登录之前可以展示的各种数据或图表的portal首页面，甚至有一些系统比较特殊在于允许接入到公司的统一单点登录机制，一旦进入到登录页面即可自动登录。对于登录之后的首页面也会存在类似的问题，我们通过在配置项当中配置对应子系统的登录页面和首页面来解决此类问题。

打开下图中的配置文件system-opts.properties



找到如下两个配置项：

#loginpage

loginpagepos=/login-form.jsp

#indexpage

indexpagepos=/index\_old.jsp

如下图：



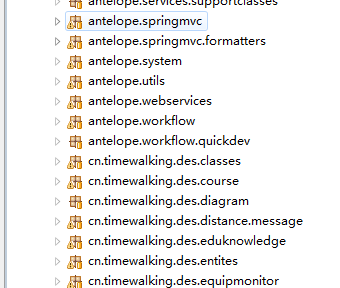
antelope框架默认使用以上两个jsp文件作为登录页和首页，请在系统webapp直接目录下创建子项目所需要的定制版登录页面和首页面，然后将其jsp文件名称按照类似的方式配置上即可。配置完毕之后，重新启动tomcat服务器以测试对应的配置是否起效。

## 项目相关的命名规范

一般项目都会有一个项目代号，如des系统 ecm系统等，一般为项目英文名称的首字母缩写等。对于项目名称的一个唯一要求就是**全小写**。

值得注意的是，虽然我们开发的是javaweb项目，但是为了提高eclipse的响应速度，减少类似于偶然引起的jsp校验等长时间等待操作，我们从svn得项目时需要以java project的项目类型得，而不是其他项目类型。

关于后台Java包，antelope平台包要以antelope为首单词命名，而子项目则需要以cn.timewalking.(项目代号).xxx命名，以和平台代码进行区分。如下图：



前台jsp代码的文件夹命名则需要以如下方式命名

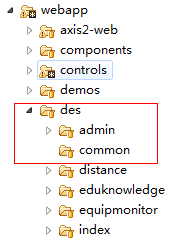
webapp

(项目代号)

(模块代号1)

(模块代号2)

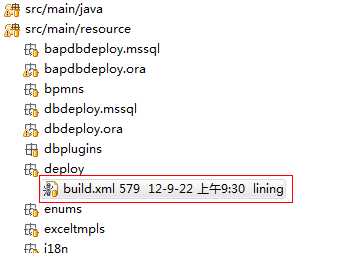
如下图，当项目代号为des时：



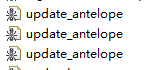
## 使用ant更新平台代码

就像之前提到的，我们使用ant来更新平台代码，使用平台代码全覆盖的方式来更新，所以需要反复提醒大家的是，对于平台代码的任意修改动作都将在更新平台代码之后被**覆盖**。

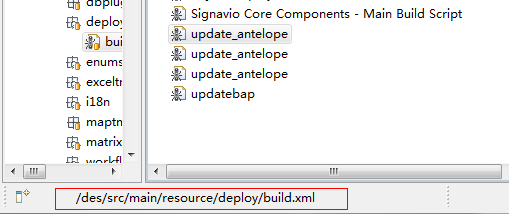
导航到如下路径/src/main/resource/deploy/build.xml，找到对应build.xml文件，如下图所示：



然后ctrl+3搜索ant将ant的视图打开，将build.xml文件拖拽到ant视图当中。ant视图当中将会出现update\_antelope的条目。由于我们的项目都统一使用antelope平台的build.xml ant配置文件，所以如果同时在开发多个项目，那么会在ant视图当中同时出现多个update\_antelope更新条目，如下图所示：



如果出现这种情况，我们可以通过eclipse下方的状态栏来具体辨别哪个update\_antelope条目属于哪个具体项目，如下图所示：



看到以des结尾则证明此更新条目为des系统的。

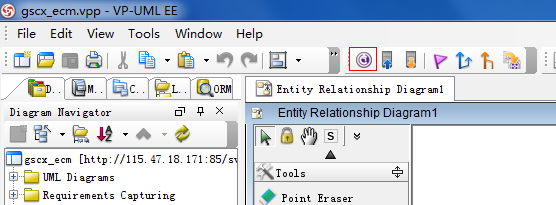
ant中出现update\_antelope条目之后，双击此条目，等待console视图中出现build success为止。

由于为全代码覆盖，所以需要在包视图中的项目节点下进行一次刷新，等待eclipse状态栏中刷新完成之后，平台代码即成功更新了。

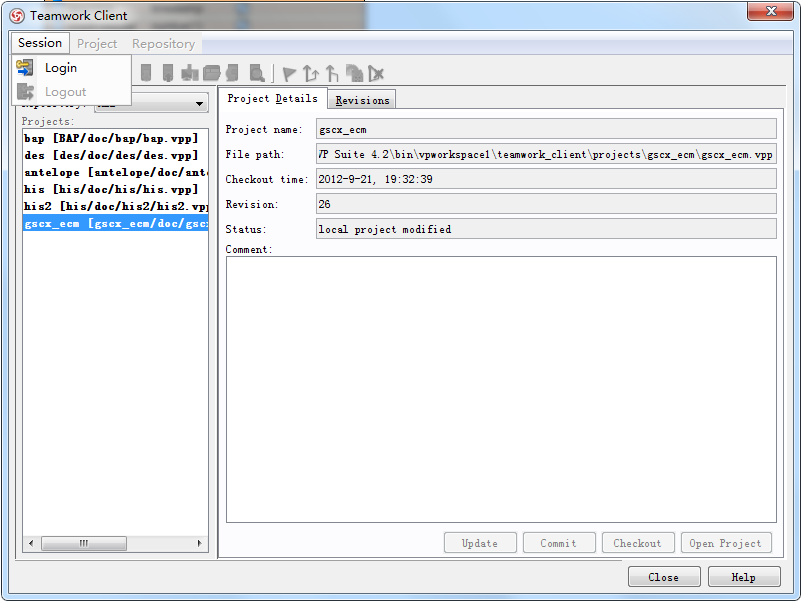
## 使用Visual Paradigm软件绘制UML

为了完成UML的统一绘制和团队合作的需要，我们使用Visual Paramdigm软件进行包括ER图在内的所有UML图形的绘制工作。

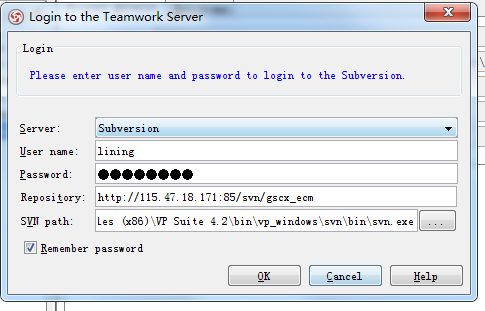
绘制UML图之前先要创建一个项目相关的UML工程，并将工程连接到svn上，首先打开Visual Paradigm for UML 7.2软件，点击工具栏中team组的 Open Teamwork Client按钮如下图：



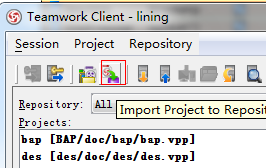
在弹出的对话框中点击login按钮:



输入对应的项目的svn目录如下



点击导入项目按钮

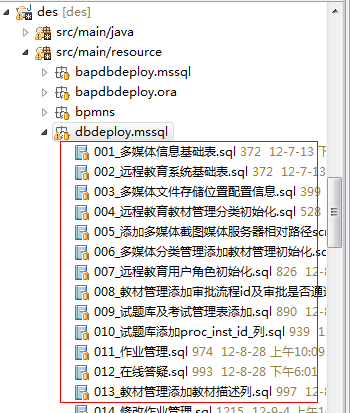


在对话框中选中当前项目的doc目录之后，使用项目代号作为Visual Paradigm项目的名称后点击确定。

在对话框中点击获取按钮和打开项目按钮即可完成与svn的连接工作。其他开发人员直接通过此方式，将VP项目连接到所开发项目的svn。

## 创建sql脚本更新文件

为了实现sql脚本的版本管理，我们使用dbdeployer开源组件对脚本的自动化升级进行了封装，当应用服务器启动时，会自动执行对应的sql升级脚本。但sql脚本的自动化依赖于对sql脚本文件的高质量维护，整齐的sql脚本目录为如下图这样的形式：



需要注意的问题有如下几个

1. 要确保sql文件保存在了dbdeploy.mssql包下（若为oracle则保存在dbdeployer.ora包下），而非bapdbdeploy.mssql下，因为bapdbdeployer.mssql为对antelope平台进行升级时需要进行升级的sql文件，是不允许存储子项目脚本文件的。
2. 要确保sql文件名称前缀的一个数字顺序，为001\_ 002\_的形式，中间不能有重复，也不能有断层，那么为了在团队合作当中确保这个顺序，**我们一定要在提交新sql脚本文件时更新当前存储脚本文件的目录，即dbdeploy.mssql目录，核对当前脚本文件前缀是否正确，如果不正确（存在重复现象）则调整正确之后再进行提交。**
3. 我们在添加更新脚本时，需要使用汉字或其他有意义的方式书写更新脚本文件名称，若有必要也可以在脚本文件当中添加必要的脚本更新注释。
4. 不要期望在已经提交了的脚本，并重启服务器已经执行了的脚本上面修改会在数据库中体现。**如果有对数据库数据进行修改的情况，一定要重新追加新的脚本文件来解决。**

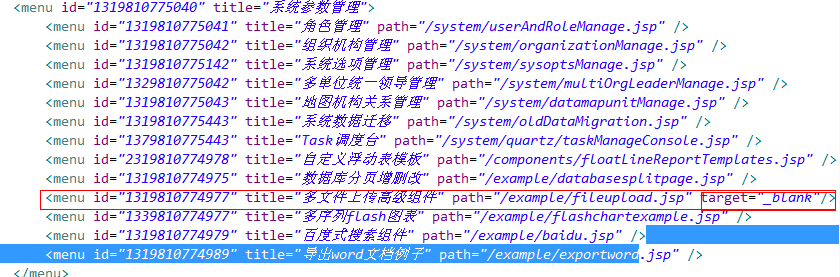
## 添加模块导航

模块导航由于要和系统权限管理进行逻辑关联，并需要完成一些系统统一处理的相关操作（如模块导航的国际化）。所以并非是程序员在任意位置都可以添加的。我们统一添加在的module-cfg.xml文件当中，对于中文而言则存在于module-cfg\_zh\_CN.xml文件当中，文件所在目录为/src/main/resource/module-cfg\_zh\_CN.xml。

模块导航在系统后台并没有限制其级数，任意级次的导航都可以添加，但是由于平台默认可以使用的导航级次为三级，并且只有第三级目录才能打开功能，所以我们将对应的功能配置到第三级目录中，如下图平台默认拥有的module-cfg\_zh\_CN.xml文件中的目录样例为例：



添加目录时需要注意如下几点：

1. 目录中的所有id都是不允许重复的，这是由于权限控制时，功能模块的权限是和功能所对应目录的id挂接到一块的。
2. 对应目录功能的链接地址不允许携带参数，如/system/abs.jsp?a=1 这样的目录是不允许的。根据经验，一般携带参数的原因是此jsp文件是具有重用性的，但是这样就带来了一个问题，如果同一个jsp文件对应两个功能模块，而同一个用户只允许访问一个功能模块，那么其实在系统级别是控制不住此用户访问另外一个功能模块的。所以此处有如此的一个限制。
3. 由于前台界面和后台的完全分离设计，那么其实前台界面得到的是一个经过权限过滤和特殊处理，并转换为json字符串的目录结构，所以如果需要对前台界面导航有一个大的变动，那么除了id、title、path、ispublic（此属性将在后面的权限控制部分详细进行介绍）这四个固定属性之外，允许任意添加属性，以供前台定制时的参数使用，例如我们可以模仿html中超链接a添加一个是否在一个新页面打开的属性target，如下:  
   

## 添加一个基本数据请求响应

使用jQuery以及springMVC创建基本的数据请求响应操作。其中除了基本的jQuery ajax和springMVC接口的使用方式之外，为了实现功能的后台权限控制，需要遵循一定的配置规则，下面来详细介绍。

首先在新建的模块jsp(也即配置到module-cfg\_zh\_CN.xml中的jsp)里面添加如下代码

<%--

基本请求响应演示

@author lining

@since 2012-09-23

--%>

<%@ page language=*"java"* pageEncoding=*"utf-8"*%>

<jsp:include page=*"/include/header2.jsp"*/>

<script>

$(**function**(){

});

</script>

<body>

</body>

</html>

下面逐一介绍这些代码的含义和书写要求。

首先是注释代码：

<%--

基本请求响应演示

@author lining

@since 2012-09-23

--%>

注释代码一般要求将jsp对应的模块功能要书写一下。然后需要添加一下创建人和创建时间。

下面一行是声明jsp页面编码行，如下

<%@ page language=*"java"* pageEncoding=*"utf-8"*%>

实际系统涉及到此行的系统行为是有所区别的，为了防止由于系统编码不统一而导致的乱码问题，系统中无论是Java代码还是jsp代码均统一使用utf-8编码。

一般jsp页面的引用文件，包括通用的js文件和css文件都是一模一样的，此外，系统会在页面加载时统一对当前页面的html控件进行初始化，为了实现代码的重用以及jsp页面的全局可控性，我们统一在此位置引用如下一行代码：

<jsp:include page=*"/include/header2.jsp"*/>

header表示引用头，2则表示当前使用的头是第二版。

再往下是一段基于jQuery框架的初始化块，如下：

<script>

$(**function**(){

});

</script>

jQuery初始化块的执行是在当前页面dom元素加载完毕开始的，所以如果需要在dom元素之前书写一段js，需要操作此dom元素，并且要在初始化jsp时执行就需要使用此初始化块的方式完成相关操作。需要说明的是，jQuery初始化块的执行是有顺序的，是按照初始化块在jsp页面出现的前后位置决定的，所以如果同一个页面有多个初始化块的话，需要注意这些代码的前后顺序，另外在header2 .jsp中所引入的js中也有初始化块，那么这些初始化块是先于所有jsp页面中书写的代码前执行的，具体可以参考jquery-ui.js的源码即可得知。

最后三行是正常的html元素，就不在赘述了。

在jsp添加了基本代码之后，我们将在初始化块当中添加获取后台数据的jQuery ajax代码，如下：

…

<script>

$(**function**(){

$.get(ctx + "/demos/index/DemoIndexController/getDemoData.vot", **function**(data){

alert(data);

});

});

</script>

…

然后，创建请求所对应的控制器类，内容如下：

**package** antelope.demos;

**import** org.springframework.stereotype.Controller;

**import** antelope.springmvc.BaseController;

@Controller

**public** **class** DemoIndexController **extends** BaseController {

}

创建此类注意两个关键的问题：

1. 必须继承自BaseController（之前提到的SingleDatagrid之所以不需要继承BaseController，是因为它本身也是BaseController的子类）
2. 必须添加@Controller的annotation,这样才能让容器发现此控制器

然后添加ajax请求所对应的方法，如下:

**package** antelope.demos;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**import** org.springframework.stereotype.Controller;

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

**import** antelope.springmvc.BaseController;

@Controller

**public** **class** DemoIndexController **extends** BaseController {

**@RequestMapping("/demos/index/DemoIndexController/getDemoData")**

**public** **void** getDemoData(HttpServletRequest req, **HttpServletResponse res**) {

}

}

注意红色的，此对象为Servlet接口中的响应对象，这个对象无论是否会被使用，在所有请求方法中都必须传递，否则请求完成之后ajax会报错。

RequestMapping中不需要添加url中的后缀.vot

在此例中，我们返回一个简单的提示字符串到前台，如下：

@RequestMapping("/demos/index/DemoIndexController/getDemoData")

**public** **void** getDemoData(HttpServletRequest req, HttpServletResponse res) **throws** IOException {

print("操作成功", res);

}

重启服务器后打开对应的功能模块，即可看到弹出的提示信息。

# 数据库数据的查询操作相关接口

最基本的数据操作都是通过BaseController类中的dao属性以及antelope.db包下的DBUtil工具类中的静态方法来完成。dao对象和DBUtil工具类的适用场合对比如下表所示：

|  |  |  |
| --- | --- | --- |
| **接口**  **对比方面** | **dao对象** | **DBUtil工具类** |
| 内部实现引擎 | Hibernate ORM引擎 | antelope自封装 |
| 实体对象的查询 | 支持 | 支持 |
| 查询无实体对象的数据列表 | 不支持 | 支持 |
| 查询单JSONObject数据 | 不支持 | 支持 |
| 查询JSONObject列表 | 不支持 | 支持 |
| 查询单值 | 不支持 | 支持 |
| 查询单值列表 | 不支持 | 支持 |
| 插入实体对象数据 | 支持 | 不支持 |
| 更新实体对象数据 | 支持 | 不支持 |
| 删除实体对象数据 | 支持 | 不支持 |
| 使用sql进行批量更新 | 支持 | 支持 |
| 查询后台分页数据 | 支持 | 支持 |

通过对比可以看出，dao更多的是和实体对象打交道，使用它来完成对对象的增删改的操作，而DBUtil更擅长于查询，无论是有实体还是无实体都如此，结果均使用JSONObject来保存，更容易将无对象对应的查询结果数据传输给前台。

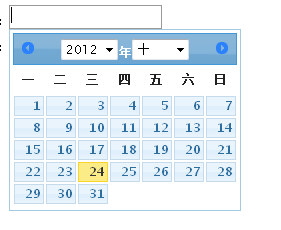
# 表单控件以及表单验证

表单控件使用jquery-ui基础表单，或者基于它的自定义扩展表单，因此若jquery-ui原生支持的表单控件均可以直接使用。若想使用它们请参照jquery-ui官方文档。本文档只介绍对jquery-ui表单控件封装或扩展后的使用方式。

框架中的表单控件主要分两种类型，一种是直接在html原生表单上面添加class完成，另一种通过初始化时书写js完成。例如，若以如下方式添加表单

<input **class=”date”**/>

则将自动添加日期控件，点击输入框时如下图：



若在代码中添加如下代码：

<script>

$(function() {

$("# myfile").fileupload("destroy").fileupload({

autosave: **true**,

enablePermanent: **false**,

filegroupsid: “abc”

});  
});

</script>

<div id=”myfile”></div>

将在此div容器内部添加如下表单



点击添加附件即可上传文件。

## 框架内置的class类型控件

|  |  |  |  |
| --- | --- | --- | --- |
| **class名称** | **控件名称及说明** | **控件外观截图** | |
| date | 日期控件，jQuery UI原生控件，优化默认添加对年月的选择，用以选择日期。 |  | |
| datetime | 日期时间控件，日期控件的扩展控件，用以同时对日期和时间进行选择。 |  | |
| button | 按钮控件，jQuery UI原生控件，用以添加jQuery UI 主题按钮，并添加按钮图标，禁用等效果接口。 |  | |
| tinymce | 基本富文本编辑框控件，用来编辑富文本，如书写文章等。 |  | |
| tabs | 页签控件，jQuery UI原生控件，用以将符合规则的dom结构加载为页签形式。  所需结构如下：  <div class=*"tabs"*>  <ul id=*"tabsul"*>  <li><a href=*"#tabs-myAll1"* >页签A</a></li>  <li><a href=*"#tabs-myAll2"* >页签B</a></li>  </ul>  <div id=*"tabs- myAll1"* class=*"tabs-myTodo"*>  <div class=*"mytodolist"*></div>  </div>  <div id=*"tabs- myAll2"* class=*"tabs-myTodo"*>  <div class=*"mytodolist"*></div>  </div>  </div> | |  | |
| ui-layout | 布局组件，jQuery布局插件，完成含有拖动边界的相关布局的初始化，所需结构如下：  <div class=*"ui-layout"*>  <div class=*"ui-layout-west"*>    </div>  <div class=*"ui-layout-center"*>    </div>  </div>  其中ui-layout必须标注，其他可选，若要想布局出更合适的布局，可以参考以下文档：http://layout.jquery-dev.net/ | |  | |
| heightadapt | *（****注意****：此css类与tinymce不可同时使用）*  自适应高度文本域在textarea上标注即可。标注了此css类的textarea具有自适应高度的功能，当输入的数据高度超过当前textarea高度时，就会自动变高。 | |  | |

## 表单验证

表单验证已经成为信息系统中必不可少的组成部分，从开发者角度讲，不进行表单验证意味着将错误的数据流入到系统中，造成程序的处理错误，过低的性能，以及显示、统计等等问题；而对于用户来说，即添加了仔细核对数据正确性的工作，也添加了辅导用户书写表单的工作。添加验证，正确地通过系统功能来指导用户填写表单，并且确保表单数据的合理性，正确性是软件开发人员义不容辞的责任。

antelope框架中为了适应各种情况，当前提供的提示方式为将控件边框变为醒目的颜色，并且当鼠标放置时提示表单的错误信息，如下：



控件的验证是通过在控件对应的html中添加对应的验证信息完成的，如：

<input required=”true”/>

表示当前的控件为必填控件。

下表列举了系统中当前已经支持的验证类型html属性：

|  |  |  |
| --- | --- | --- |
| **表单域控件中可书写的属性** | **可书写的属性值** | **验证功能说明** |
| required | true | 必填项 提示：请填写此字段  *注意：流程全界面组件中的暂存不受此验证属性限制。* |
| maxlength2 | int 型数值 | 最大长度限制，由于对于普通文本框来说maxlength已经允许直接限制长度，但在有些情况下需要允许用户输入过长字符串但给与提示的形式告诉用户过长，所以此处改用maxlength2来完成。 |
| num | true | 必须为数字类型，包括整形和浮点型。如12.34  *注意：此属性不能限制必填* |
| int | true | 必须为整数类型，如12  *注意：此属性不能限制必填* |
| max | int 或 float 类型值 | 限制填写的最大值（包含） |
| min | int 或 float 类型值 | 限制填写的最小值（包含） |
| after | 填写与当前表单域值对比的表单域的名称：name属性值 | 用户验证某日期是否在所对比日期之后，也不允许相等，否则给与提示 |
| notBefore | 填写与当前表单域值对比的表单域的名称：name属性值 | 用户验证某日期不能在所对比日期之前，但可以相当，否则给与提示。 |

对于表单来说，验证属性可以同时添加多个，但会按照上表的顺序依次校验，校验表单时会同时校验表单中的所有表单域，若存在多个表单不符合验证则均会给与提示。

# 计划任务

数据定时处理，定时获取、更新。定时发邮件，调用WebService接口，在很多情况下计划任务成为了应用程序不可或缺的功能。

计划任务要求不仅仅是可以写在程序里面的，而且可以由用户或者管理员进行配置执行，现在这些功能都已经在antelope中稳定地集成了，而且使用spring的方式更提高了其灵活性。

当前系统中支持两种计划任务启用方式，一种是spring所携带的annotation注解的方式启用，另一种是通过实现接口启用，其中annotation注解方式简单直观，但功能不够灵活，实现接口的方式稍复杂，但灵活性更高，允许用户可配。

两种方式相同的是，都使用cron表达式来完成执行时间的配置。

## Spring注解方式

@Scheduled(cron="0 30 23 ? \* \*")

public void sendEmailForCaseManageLitigation() throws SQLException, Exception {

｝

通过@Scheduled(cron="0 30 23 ? \* \*")即可完成计划任务的配置，方法会按照参数中的cron表达式定期执行。需要注意的是，这个方法所在的类必须为spring容器管理的类，也即必须存在@Service @Controller @Component注解其中之一标注在此类上。

## 接口方式

package antelope.demos;

@Component

public class ScheduleTaskDemo implements TaskScheduleService {

@Override

@CronExpression("0 49 21 ? \* \*")

public void execute(JobExecutionContext arg0) throws JobExecutionException {

System.out.print("分大方撒旦");

}

@Override

public TaskGroup getTaskGroupName() {

return TaskGroup.DEFAULT;

}

}

以上代码即为用户可配置的接口方式计划任务，需要实现antelope.quartz.TaskScheduleService接口，并实现其中的execute方法，方法上方添加cron表达式的注解，此注解为在用户自行配置之前所使用的默认表达式，同样需要注意实现完成之后需要将此类标注为spring所管理的类。

**注意：**虽然标注为了spring所管理的类，但计划任务所执行的类实例并不在当前spring所管理下，所以如果希望使用SpringUtils所管理的类，如BaseComponent中常用的方法或对象，需要通过SpringUtils.getBeans等实用方法进行获取。如下获取dao的方式：

@Override

@CronExpression("0 49 21 ? \* \*")

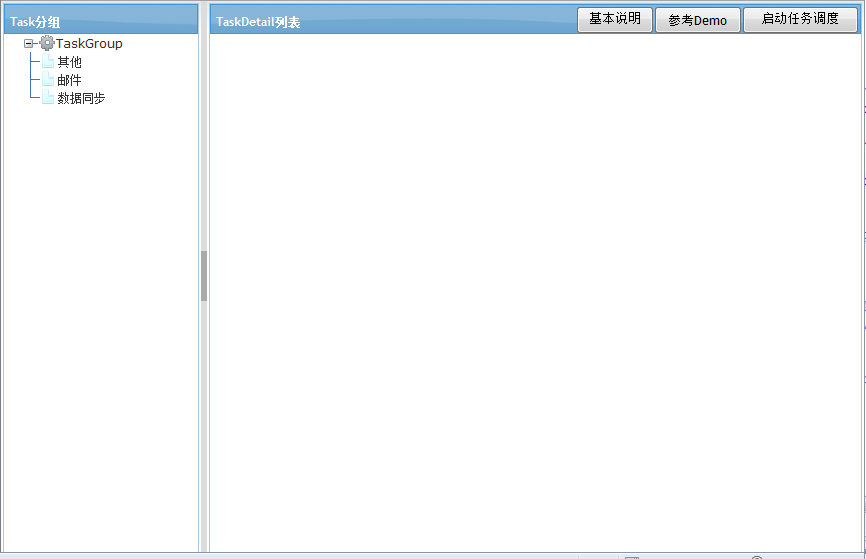
public void execute(JobExecutionContext arg0) throws JobExecutionException {

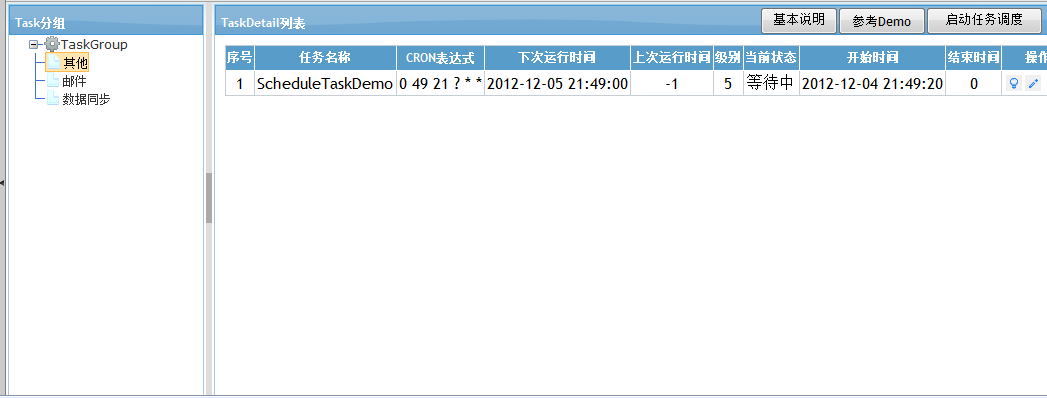
JPABaseDao dao = SpringUtils.getBean(JPABaseDao.class);

}

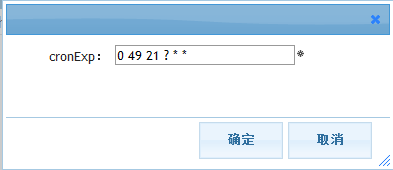
实现getTaskGroupName方法，将会把此类的计划任务放到对应的分组下，具体可查看TaskGroup枚举类表达的当前所拥有的分组类型。

实现完成之后，进入系统的Task调度台功能，如下图：



此时计划任务并未启动，点击右上角的按钮，提示完成之后出现一条记录：

列表中将显示与计划任务有关的数据，计划任务也同时启动了，点击操作列中的修改按钮即可打开如下对话框修改cron表达式：



通过它来完成cron表达式的修改，修改完成之后新表达式会立刻起效。

若对cron表达式不熟悉，可点右上角的 （基本说明） 与 （参考demo） 按钮，其中有一些cron表达式的语法说明以及简单demo。

# 附录 框架主要接口API

## JPABaseDao

系统通用Dao基础类 完成系统常用的Dao相关操作

|  |  |  |  |
| --- | --- | --- | --- |
| **开放使用的属性或方法名** | **方法描述** | **参数说明** | **返回值说明** |
| +getAll(clazz : Class<A>) : java.util.List<A> | 获取实体类所对应表中的所有数据 | clazz : Class<A>  实体类类型对象 | java.util.List<A>  实体对象列表或含有0个元素的空列表 |
| +getBy(sid : Object, clazz : Class<A>) : A | 根据主键获取实体表中的类 | sid : Object  实体对应主键id  clazz : Class<A>  实体类类型对象 | A  单个实体对象或null（未找到） |
| +getBy(dbcolname : String, value : Object, clazz : Class<A>) : java.util.List<A> | 根据单个外键获取实体表中的类对象 | dbcolname : String  对应外键数据库表列名  value : Object  外键值  clazz : Class<A>  实体类类型对象 | java.util.List<A>  实体对象列表或含有0个元素的空列表 |
| +query(sql : String, resultClass : Class<A>) : java.util.List<A> | 根据sql获取实体List对象 | sql : String  sql字符串  resultClass : Class<A>  实体类类型对象 | java.util.List<A>  实体对象列表或含有0个元素的空列表 |
| +querySingle(sql : String, resultClass : Class<A>) : A | 根据sql获取单个实体对象值，若不存在则返回null | sql : String  sql字符串  resultClass : Class<A>  实体类对象 | A  单个实体对象或null（未找到） |
| +query(sql : String, params : Object [], resultClass : Class<A>) : java.util.List<A> | 根据sql获取实体List对象,允许传递一个参数数组 | sql : String  sql字符串  params : Object []  sql中问号所标识的参数  resultClass : Class<A>  实体类类型对象 | java.util.List<A>  实体对象列表或含有0个元素的空列表 |
| +querySingle(sql : String, params : Object [], resultClass : Class<A>) : A | 根据sql获取单个实体对象值，允许传递一个参数数组 | sql : String  sql字符串  params : Object []  sql中问号所标识的参数  resultClass : Class<A>  实体类类型对象 | A  单个实体对象或null（未找到） |
| +query(sql : String, param : Object, resultClass : Class<A>) : java.util.List<A> | 根据sql获取单个实体对象值，允许传递一个单值参数 | sql : String  sql字符串  params : Object  sql中问号所标识的参数  resultClass : Class<A>  实体类类型对象 | java.util.List<A>  实体对象列表或含有0个元素的空列表 |
| +querySingle(sql : String, param : Object, resultClass : Class<A>) : A | 根据sql获取单个实体对象值，允许传递一个单值参数 | sql : String  sql字符串  params : Object  sql中问号所标识的参数  resultClass : Class<A>  实体类类型对象 | A  单个实体对象或null（未找到） |
| +query(sql : String, params : java.util.List<Object>, resultClass : Class<A>) : java.util.List<A> | 根据sql获取单个实体对象值，允许传递一个参数列表 | sql : String  sql字符串  params : java.util.List<Object>  sql中问号所标识的参数列表  resultClass : Class<A>  实体类类型对象 | java.util.List<A>  实体对象列表或含有0个元素的空列表 |
| +querySingle(sql : String, params : java.util.List<Object>, resultClass : Class<A>) : A | 根据sql获取单个实体对象值，允许传递一个参数列表 | sql : String  sql字符串  params : java.util.List<Object>  sql中问号所标识的参数列表  resultClass : Class<A>  实体类类型对象 | A  单个实体对象或null（未找到） |
| +query(sql : String, params : Object [], resultClass : Class<?>, pageParams : antelope.utils.PageParams) : antelope.utils.PageItem | 根据sql获取以及分页参数，获取保存对象的单页数据，允许传递一个参数数组。 | sql : String  sql字符串  params : Object []  sql中问号所标识的参数  resultClass : Class<A>  实体类类型对象  pageParams : antelope.utils.PageParams  后台分页参数对象 | antelope.utils.PageItem  单页数据对象 |
| +query(sql : String, params : java.util.List<Object>, resultClass : Class<?>, pageParams : antelope.utils.PageParams) : antelope.utils.PageItem | 根据sql获取以及分页参数，获取保存对象的单页数据，允许传递一个参数列表。 | sql : String  sql字符串  params : java.util.List<Object>, sql中问号所标识的参数  resultClass : Class<A>  实体类类型对象  pageParams : antelope.utils.PageParams  后台分页参数对象 | antelope.utils.PageItem  单页数据对象 |
| +query(sql : String, param : Object, resultClass : Class<?>, pageParams : antelope.utils.PageParams) : antelope.utils.PageItem | 根据sql获取以及分页参数，获取保存对象的单页数据，允许传递一个单值参数。 | sql : String  sql字符串  param : Object  sql中问号所标识的参数  resultClass : Class<A>  实体类类型对象  pageParams : antelope.utils.PageParams  后台分页参数对象 | antelope.utils.PageItem  单页数据对象 |
| +query(sql : String, resultClass : Class<?>, pageParams : antelope.utils.PageParams) : antelope.utils.PageItem | 根据sql获取以及分页参数，获取保存对象的单页数据。 | sql : String  sql字符串  resultClass : Class<A>  实体类类型对象  pageParams : antelope.utils.PageParams  后台分页参数对象 | antelope.utils.PageItem  单页数据对象 |
| +batchInsertOrUpdate(beanList : java.util.List<A>) : void | 对数据进行批量更新或修改 | beanList : java.util.List<A>  要保存的实体类对象列表 | void |
| +insertOrUpdate(instance : A) : void | 添加新项目或者更新已有项目，若根据id或者联合id找到项目则更新，否则添加一个新项目进去 | instance : A  要保存的实体类对象 | void |
| +updateBySQL(sql : String, params : Object []) : int | 执行sql批量更新数据库数据，允许传递一个参数数组 | sql : String  sql字符串  params : Object []  sql中问号所标识的参数 | int  影响的数据行数 |
| +updateBySQL(sql : String, param : Object) : int | 执行sql批量更新数据库数据，允许传递一个单值参数 | sql : String  sql字符串  params : Object  sql中问号所标识的参数 | int  影响的数据行数 |
| +updateBySQL(sql : String) : int | 执行sql批量更新数据库数据 | sql : String  sql字符串 | int  影响的数据行数 |
| +updateBySQL(sql : String, params : java.util.List<Object>) : int | 执行sql批量更新数据库数据，允许传递一个参数列表 | sql : String  sql字符串  params : java.util.List<Object>  sql中问号所标识的参数 | int  影响的数据行数 |
| +deleteBy(key : Object, clazz : Class<A>) : int | 根据实体主键删除数据 | key : Object  要删除的实体对象主键值  clazz : Class<A>  实体类类型对象 | int  影响的数据行数 |
| +remove(obj : Object) : void | 删除实体对象 | obj : Object  实体对象 | void |

## DBUtil

数据库查询工具类

|  |  |  |  |
| --- | --- | --- | --- |
| **开放使用的属性或方法名** | **方法描述** | **参数说明** | **返回值说明** |
| +query(sql : String, param : Object) : java.util.List<java.util.Map<String, Object>> | 根据sql查询一个储存数据的Map列表，允许传递一个单值参数 | sql : String  sql字符串  param : Object  sql中问号所标识的参数 | java.util.List<java.util.Map<String, Object>>  储存结果数据的Map列表，若未查询到，则返回含有0个元素的空列表 |
| +executeUpdate(sql : String) : boolean | 执行一个sql，完成对数据库数据的批量更新操作。 | sql : String  sql字符串 | boolean  返回所影响的行数是否大于0 |
| +executeUpdate(conn : java.sql.Connection, sql : String) : boolean | 执行一个sql，完成对数据库数据的批量更新操作，允许传递一个数据库连接，注意此方法传入的数据库连接将不被包含在当前事务当中。 | conn : java.sql.Connection  更新时所使用的数据库连接。  sql : String  sql字符串 | boolean  返回所影响的行数是否大于0 |
| +executeUpdate(conn : java.sql.Connection, sql : String, params : Object []) : boolean | 执行一个sql，完成对数据库数据的批量更新操作，允许传递一个数据库连接和一个参数数组，注意此方法传入的数据库连接将不被包含在当前事务当中。 | conn : java.sql.Connection  更新时所使用的数据库连接。  sql : String  sql字符串  params : Object []  sql中问号所标识的参数 | boolean  返回所影响的行数是否大于0 |
| +query(sql : String, params : Object []) : java.util.List<java.util.Map<String, Object>> | 根据sql查询一个储存数据的Map列表，允许传递一个参数数组 | sql : String  sql字符串  param : Object[]  sql中问号所标识的参数 | java.util.List<java.util.Map<String, Object>>  储存结果数据的Map列表，若未查询到，则返回含有0个元素的空列表 |
| +query(conn : java.sql.Connection, sql : String, params : Object []) : java.util.List<java.util.Map<String, Object>> | 根据sql查询一个储存数据的Map列表，允许传递一个参数列表和一个数据库连接，注意此连接不会参与到当前的事务当中 | conn : java.sql.Connection  查询时所使用的数据库连接。  sql : String  sql字符串  params : Object []  sql中问号所标识的参数 | java.util.List<java.util.Map<String, Object>>  储存结果数据的Map列表，若未查询到，则返回含有0个元素的空列表 |
| +queryJSON(sql : String, params : Object []) : java.util.List<antelope.utils.JSONObject> | 根据sql查询一个储存数据的JSONObject列表，允许传递一个参数数组 | sql : String  sql字符串  params : Object  sql中问号所标识的参数 | java.util.List<antelope.utils.JSONObject>  储存结果数据的JSONObject列表，若未查询到，则返回含有0个元素的空列表 |
| +getSession() : org.hibernate.Session | 获取hibernate的Session | 无 | 当前系统启动后所使用的hibernate的Session |
| +queryEntities(sql : String, params : Object [], klass : Class<A>) : java.util.List<A> | 根据sql查询一个存储实体数据的列表，允许传递一个参数数组和一个实体类型类 | sql : String  sql字符串  params : Object  sql中问号所标识的参数  klass : Class<A>  实体类类型对象 | java.util.List<A>  储存结果数据的实体对象A的列表，若未查询到，则返回含有0个元素的空列表 |
| +queryEntities(sql : String, klass : Class<A>) : java.util.List<A> | 根据sql查询一个存储实体数据的列表，允许传递一个实体类型类 | sql : String  sql字符串  klass : Class<A>  实体类类型对象 | java.util.List<A>  储存结果数据的实体对象A的列表，若未查询到，则返回含有0个元素的空列表 |
| +queryEntities(sql : String, params : java.util.List<Object>, klass : Class<A>) : java.util.List<A> | 根据sql查询一个存储实体数据的列表，允许传递一个参数列表和一个实体类型类 | sql : String  sql字符串  params : java.util.List<Object>  sql中问号所标识的参数  klass : Class<A>  实体类类型对象 | java.util.List<A>  储存结果数据的实体对象A的列表，若未查询到，则返回含有0个元素的空列表 |
| +queryEntities(sql : String, param : Object, klass : Class<A>) : java.util.List<A> | 根据sql查询一个存储实体数据的列表，允许传递一个单值参数和一个实体类型类 | sql : String  sql字符串  param : Object  sql中问号所标识的参数  klass : Class<A>  实体类类型对象 | java.util.List<A>  储存结果数据的实体对象A的列表，若未查询到，则返回含有0个元素的空列表 |
| +getColumnNameSet(rs : java.sql.ResultSet) : java.util.Set<String> | 根据查询结果集获取查询结果列名集合 | rs : java.sql.ResultSet  查询结果集 | java.util.Set<String>  列名集合 |
| +wrapToEntity(rs : java.sql.ResultSet, bean : A, columnset : java.util.Set<String>) : A | 将查询结果集数据根据列名集合封装到一个对象当中 | rs : java.sql.ResultSet  查询结果集  bean : A  需要封装的对象  columnset : java.util.Set<String>  查询结果集列名集合 | A  封装完成的对象 |
| +queryJSON(sql : String, param : Object) : java.util.List<antelope.utils.JSONObject> | 根据sql查询一个储存数据的JSONObject列表，允许传递一个单值参数 | sql : String  sql字符串  params : Object  sql中问号所标识的参数 | java.util.List<antelope.utils.JSONObject>  储存结果数据的JSONObject列表，若未查询到，则返回含有0个元素的空列表 |
| +queryJSON(sql : String, params : java.util.List<Object>) : java.util.List<antelope.utils.JSONObject> | 根据sql查询一个储存数据的JSONObject列表，允许传递一个参数列表 | sql : String  sql字符串  params : java.util.List<Object>  sql中问号所标识的参数 | java.util.List<antelope.utils.JSONObject>  储存结果数据的JSONObject列表，若未查询到，则返回含有0个元素的空列表 |
| +queryJSON(conn : java.sql.Connection, sql : String, params : Object []) : java.util.List<antelope.utils.JSONObject> | 根据sql查询一个储存数据的JSONObject列表，允许传递一个参数数组和一个外部数据库连接 | conn : java.sql.Connection  外部数据库连接  sql : String  sql字符串  params : java.util.List<Object>  sql中问号所标识的参数 | java.util.List<antelope.utils.JSONObject>  储存结果数据的JSONObject列表，若未查询到，则返回含有0个元素的空列表 |
| +queryJSON(sql : String) : java.util.List<antelope.utils.JSONObject> | 根据sql查询一个储存数据的JSONObject列表 | sql : String  sql字符串 | java.util.List<antelope.utils.JSONObject>  储存结果数据的JSONObject列表，若未查询到，则返回含有0个元素的空列表 |
| +querySingleJSON(sql : String) : antelope.utils.JSONObject | 根据sql查询一个储存数据的JSONObject单对象 | sql : String  sql字符串 | antelope.utils.JSONObject  储存结果数据的JSONObject对象，若未查询到，则返回null |
| +querySingleJSON(sql : String, params : Object []) : antelope.utils.JSONObject | 根据sql查询一个储存数据的JSONObject单对象，允许传递一个参数数组 | sql : String  sql字符串  params : Object []  sql中问号所标识的参数 | antelope.utils.JSONObject  储存结果数据的JSONObject对象，若未查询到，则返回null |
| +querySingleJSON(sql : String, param : Object) : antelope.utils.JSONObject | 根据sql查询一个储存数据的JSONObject单对象，允许传递一个单值参数 | sql : String  sql字符串  param : Object  sql中问号所标识的参数 | antelope.utils.JSONObject  储存结果数据的JSONObject对象，若未查询到，则返回null |
| +querySingleJSON(sql : String, params : java.util.List<Object>) : antelope.utils.JSONObject | 根据sql查询一个储存数据的JSONObject单对象，允许传递一个参数列表 | sql : String  sql字符串  params : java.util.List<Object>  sql中问号所标识的参数 | antelope.utils.JSONObject  储存结果数据的JSONObject对象，若未查询到，则返回null |
| +query(sql : String, params : java.util.List<Object>) : java.util.List<java.util.Map<String, Object>> | 根据sql查询一个储存数据的Map列表，允许传递一个参数列表 | sql : String  sql字符串  params : java.util.List<Object>  sql中问号所标识的参数 | java.util.List<java.util.Map<String, Object>>  储存结果数据的Map列表，若未查询到，则返回含有0个元素的空列表 |
| +query(sql : String) : java.util.List<java.util.Map<String, Object>> | 根据sql查询一个储存数据的Map列表 | sql : String  sql字符串 | java.util.List<java.util.Map<String, Object>>  储存结果数据的Map列表，若未查询到，则返回含有0个元素的空列表 |
| +querySingleValList(sql : String) : java.util.List<Object> | 根据sql查询一个储存数据的单值列表 | sql : String  sql字符串 | java.util.List<Object> 储存结果数据的单值列表，若未查询到，则返回含有0个元素的空列表 |
| +querySingleValList(sql : String, params : Object [], datatype : DataType) : java.util.List<Object> | 根据sql查询一个储存数据的单值列表，允许指定单值的数据类型，并允许传递一个参数数组 | sql : String  sql字符串  params : Object [] sql中问号所标识的参数  datatype : DataType  返回值的数据类型枚举  可选值：Integer, BigDecimal, Long, String, Double, Timestamp | java.util.List<Object> 储存结果数据的单值列表，若未查询到，则返回含有0个元素的空列表 |
| +querySingleValList(sql : String, param : Object, datatype : DataType) : java.util.List<Object> | 根据sql查询一个储存数据的单值列表，允许指定单值的数据类型，并允许传递一个单值参数 | sql : String  sql字符串  params : Object  sql中问号所标识的参数  datatype : DataType  返回值的数据类型枚举  可选值：Integer, BigDecimal, Long, String, Double, Timestamp | java.util.List<Object> 储存结果数据的单值列表，若未查询到，则返回含有0个元素的空列表 |
| +querySingleValList(sql : String, params : java.util.List<Object>, datatype : DataType) : java.util.List<Object> | 根据sql查询一个储存数据的单值列表，允许指定单值的数据类型，并允许传递一个参数列表 | sql : String  sql字符串  params : java.util.List<Object>  sql中问号所标识的参数  datatype : DataType  返回值的数据类型枚举  可选值：Integer, BigDecimal, Long, String, Double, Timestamp | java.util.List<Object> 储存结果数据的单值列表，若未查询到，则返回含有0个元素的空列表 |
| +querySingleVal(sql : String, datatype : DataType) : Object | 根据sql查询一个单值数据，允许指定单值的数据类型 | sql : String  sql字符串  datatype : DataType  返回值的数据类型枚举  可选值：Integer, BigDecimal, Long, String, Double, Timestamp | Object  保存单值的单值对象，若未查询到则返回null |
| +querySingleString(sql : String, params : Object []) : String | 根据sql查询一个单字符串对象，允许传递一个参数数组 | sql : String  sql字符串  params : Object []  sql中问号所标识的参数 | String  保存单字符串值对象，若未查到则返回null |
| +querySingleVal(sql : String, params : Object [], datatype : DataType) : Object | 根据sql查询一个单值数据，允许指定单值的数据类型，并且允许传递一个参数数组 | sql : String  sql字符串  params : Object []  sql中问号所标识的参数 | Object  保存单值的单值对象，若未查询到则返回null |
| +queryCount(sql : String, params : Object []) : int | 根据sql查询一个单整形数据，允许传递一个参数数组 | sql : String  sql字符串  params : Object []  sql中问号所标识的参数 | int  返回的单整形数据，若未查到，则会报错。 |
| +queryStrings(sql : String, params : Object []) : java.util.List<String> | 根据sql查询一个存储字符串的列表，允许传递一个参数数组 | sql : String  sql字符串  params : Object []  sql中问号所标识的参数 | java.util.List<String>  保存结果的字符串列表，若未查询到，则返回含有0个元素的空列表 |
| +queryStrings(sql : String) : java.util.List<String> | 根据sql查询一个存储字符串的列表 | sql : String  sql字符串 | java.util.List<String>  保存结果的字符串列表，若未查询到，则返回含有0个元素的空列表 |
| +queryStrings(sql : String, param : Object) : java.util.List<String> | 根据sql查询一个存储字符串的列表，允许传递一个单值参数 | sql : String  sql字符串  param : Object  sql中问号所标识的参数 | java.util.List<String>  保存结果的字符串列表，若未查询到，则返回含有0个元素的空列表 |
| +queryCount(sql : String, param : Object) : int | 根据sql查询一个单整形数据，允许传递一个单值参数 | sql : String  sql字符串  param : Object  sql中问号所标识的参数 | int  返回的单整形数据，若未查到，则会报错。 |
| +querySingleVal(sql : String, param : Object, datatype : DataType) : Object | 根据sql查询一个单值数据，允许指定单值的数据类型，并且允许传递一个单值参数 | sql : String  sql字符串  param: Object  sql中问号所标识的参数  datatype : DataType  返回值的数据类型枚举  可选值：Integer, BigDecimal, Long, String, Double, Timestamp | Object  保存单值的单值对象，若未查询到则返回null |
| +querySingleVal(sql : String, params : java.util.List<Object>, datatype : DataType) : Object | 根据sql查询一个单值数据，允许指定单值的数据类型，并且允许传递一个参数列表 | sql : String  sql字符串  param: Object  sql中问号所标识的参数  datatype : DataType  返回值的数据类型枚举  可选值：Integer, BigDecimal, Long, String, Double, Timestamp | Object  保存单值的单值对象，若未查询到则返回null |
| +queryJSON(sql : String, params : java.util.List<Object>, pageparams : antelope.utils.PageParams) : antelope.utils.PageItem | 根据sql查询一个储存数据的JSONObject单页数据，需要传递一个分页参数和一个参数列表 | sql : String  sql字符串  params : java.util.List<Object>  sql中问号所标识的参数  pageparams : antelope.utils.PageParams  分页参数 | antelope.utils.PageItem  储存结果数据的JSONObject单页数据对象 |
| +queryJSON(sql : String, param : Object, pageparams : antelope.utils.PageParams) : antelope.utils.PageItem | 根据sql查询一个储存数据的JSONObject单页数据，需要传递一个分页参数和一个单值参数 | sql : String  sql字符串  param : Object  sql中问号所标识的参数  pageparams : antelope.utils.PageParams  分页参数 | antelope.utils.PageItem  储存结果数据的JSONObject单页数据对象 |
| +queryJSON(sql : String, params : Object [], pageparams : antelope.utils.PageParams) : antelope.utils.PageItem | 根据sql查询一个储存数据的JSONObject单页数据，需要传递一个分页参数和一个参数数组 | sql : String  sql字符串  params : Object[]  sql中问号所标识的参数  pageparams : antelope.utils.PageParams  分页参数 | antelope.utils.PageItem  储存结果数据的JSONObject单页数据对象 |