**NSP 开 发 指 南**

编写：

校对：

审核： 年 月 日

标审： 年 月 日

批准： 年 月 日

郑州新开普电子股份有限公司

修订记录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 日期 | 修订版本 | 描述 | 作者 | 审核人 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

目录

[1. 概述 3](#_Toc378340292)

[2. 开发架构说明 4](#_Toc378340293)

[2.1 构建组件化场景和思路描述 5](#_Toc378340294)

[2.2 请求处理链适合场景 6](#_Toc378340295)

[2.3 服务Resource处理链 7](#_Toc378340296)

[2.4 开发架构机制描述 10](#_Toc378340297)

[2.1.1 对象通用封装 11](#_Toc378340298)

[2.1.2 请求参数数据绑定 18](#_Toc378340299)

[2.1.3 注解式控制器的数据验证、类型转换及格式化 22](#_Toc378340300)

[2.1.4 服务消息对象封装 31](#_Toc378340301)

[2.1.5 快速查询通道 33](#_Toc378340302)

[2.1.6 异常处理 36](#_Toc378340303)

[2.1.7 系统异常断言 39](#_Toc378340304)

[2.1.8 视图地址统一管理： 40](#_Toc378340305)

[3. 开发实例说明 42](#_Toc378340306)

# 概述

NSP开发框架用以支持公司的产品与项目建设，其基于RESTLET、Spring、Hibernate开源框架，完成了请求处理层、业务逻辑层及持久化层的架构封装。同时框架提供了对JSP、Velocity等前端模板技术的支持。并且基于项目构建工具MAVEN，实现了构建期组件化。

开发框架志在提供一种轻量级的开发模式，同时兼顾可配置性的实现。通过对请求参数的封装、查询结果的通用化封装等手段，简化了程序员的开发工作，使其专注与业务逻辑建设。

对于日志、缓冲等功能，框架基于组件化的方式（插件）提供，在产品建设与项目建设需要时，通过MAVEN配置对相应插件的依赖性就可以直接使用，在项目打包时，MAVEN会自动进行依赖性分析，并完成打包工作。这种基于构建期组件化的方式，可以支持公司业务建设的大部分场景。

# 开发架构说明

公司涉及的业务领域与项目规模存在较大差异。同时随着市场的拓展，智慧城市、智慧校园、O2O等概念的落地，对大数据业务的应对能力也提到建设议程，应用提供一个什么样的架构来应对这些挑战是当前的一个核心问题。

云计算的概念来源于互联网，发展自分布式计算，其编程模型注重高效的远程接口实现。传统行业主要是企业计算模型，其更注重系统稳定性、可配置性与用户界面的友好性。为了同时满足这两方面需求，在架构层面的决策本身就是存在矛盾的。

为此，NSP开发架构提出一种“微内核”的思路解决这个问题。首先以开发架构封装一个包含“请求处理”+“业务处理”+“持久化”的“微内核”，其应用即适合于企业计算的业务建设，同时适合与高效远程接口的实现。其次，围绕开发框架，建设一系列的插件（例如：日志插件、缓存插件、NoSQL插件等）。当开发不同的类型的产品与项目时，可以选择不同插件。对这些插件的依赖性管理，都通过MAVEN来实现。

对于每个插件的建设也是独立与灵活的。以缓存插件为例，可以定义一致化的缓存接口，但是可以基于concurrentHashmap实现本地缓存，也可以基于memcache实现集中式分布式缓冲，甚至可以基于Redis实现更高效的分布式缓存。对于接口与实现的对应在插件内部完成，在产品与项目研发中使用插件时，可以针对不同的场景进行不同缓冲策略的选择。同时，对于开发人员来说，不需要了解这些差异性，他们面对统一的缓冲接口编程，实现自己的业务逻辑。项目部署过程中，只要更改相应的配置下，就可以适应单例部署或集群部署的场景。

## 构建组件化场景和思路描述

为了实现“大项目要做好，小项目要做快”的目标，首先要认识到针对不同类型的项目，由于其复杂性、规模性、用户特点等方面的不同，实际上需要采用不同的架构实现，对产品建设是一个道理。所以，NSP摒弃统一个架构支持所有产品、项目建设的思路，而是基于构建期（设计、开发、测试、发布）的组件化，实现针对不同产品、项目可以灵活组合不同的架构来实现。

统一架构支持所有产品与项目建设会造成很多问题。由于这个定位，会造成整体架构必须支持最复杂的产品建设与项目建设，从而对整体架构层次化、组件划分、配置模型等方面的设计都必须按照最复杂的场景来设计，从而造成整体架构庞大、臃肿、配置繁杂。但是对于实际的产品与项目建设来说，这样的架构实际上是弊大于利的。90%的项目可能只需要架构提供的10%~30%的灵活性，但是为实现多于灵活性，会对设计、开发、测试、部署、升级带来多出一倍甚至三倍的成本。

为了解决上述问题NSP平台建设思路是：重视构建期组件化建设，以设计与实践驱动相结合的方式推进公司产品的整体组件化。以开发框架建设统一公司的开发模式，作为产品与项目建设的“微内核”，能够灵活与扩展应用，借助MAVEN对软件项目的预处理、编译、打包、测试、分发的构建期活动进行全面支持。优秀的依赖管理系统更是为软件组件解决构建期依赖性问题提供了优秀的解决方案。

## 请求处理链适合场景

从用户端客户发送三类请求：静态资源的请求；rest资源的请求；MultiView控制器的请求。对于静态资源的请求，由Web容器直接返回给客户端，这是一个最高效的模式。对于另外两种请求的处理说明如下：

1、REST资源的请求：这条请求路径更适合高效远程接口的封装。请求由Restlet提供了Servlet接受，进入Restlet路由与处理链条，框架实现了对前各种类型请求参数的封装，并将请求最终导向目标的Restlet资源对象。

资源对象调用业务服务对象完成业务逻辑。业务服务对象调用数据访问对象完成持久化数据的访问，调用依赖组件完成业务逻辑，也可能调用依赖组件的远程接口完成业务逻辑。

2、MultiView资源的请求：这条请求路径适合输出人机交互的UI界面，不适合高效远程接口的封装。这条处理路径支持JSP、Velocity、Freemarker等通用页面技术，适合重用以前的工作成果，开发者可以选用熟悉的技术进行项目建设。请求由controller的servlet接受，进入controller的路由与处理链条，框架实现了对前各种类型请求参数的封装，并将请求最终导向目标的MultiView资源对象。

## 服务Resource处理链

对基于Restlet资源的请求处理过程进行详细说明：

* 在web容器配置文件中进行如下配置，将以/restful/为前缀的url导向restlet组件进行处理

<servlet>

<servlet-name>restlet</servlet-name> <servlet-class>com.noelios.restlet.ext.spring.SpringServerServlet</servlet-class>

<init-param>

<param-name>org.restlet.component</param-name>

<param-value>component</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>restlet</servlet-name>

<url-pattern>/restful/\*</url-pattern>

</servlet-mapping>

* config/restlet-conf/restletContext.xml文件中的如下配置，指定了处理请求的Component，以及默认的Application,配置总路由。

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi=<http://www.w3.org/2001/XMLSchema-instance> xmlns:util="http://www.springframework.org/schema/util"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

<bean id="component.context"

class="org.springframework.beans.factory.config.PropertyPathFactoryBean" />

<bean id="exceptionStatusService"

class="cn.newcapec.framework.core.exception.ExceptionStatusService">

</bean>

<bean id="component" class="org.restlet.ext.spring.SpringComponent">

<property name="defaultTarget" ref="application" />

</bean>

<bean id="application" class="cn.newcapec.framework.core.rest.BaseApplication">

<constructor-arg ref="component.context" />

<property name="root" ref="restRoute" />

<property name="statusService" ref="exceptionStatusService" />

</bean>

<bean id="restRoute" class="org.restlet.ext.spring.SpringRouter">

<property name="attachments">

<map>

<!-- 门户接口 -->

<entry key="/portalProxyService" value-ref="portalRoute" />

</map>

</property>

</bean>

</beans>

* config/{组件包名}/resetlet-conf/组件名Context.xml中，配置的二级路由与Restlet资源类的映射关系

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:util="http://www.springframework.org/schema/util"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

<bean id="portalRoute" class="org.restlet.ext.spring.SpringRouter">

<property name="attachments">

<map>

<!--office模式资源 -->

<entry key="/office/{method}">

<bean class="org.restlet.ext.spring.SpringFinder">

<lookup-method name="createResource" bean="officeResource" />

</bean>

</entry>

</map>

</property>

</bean>

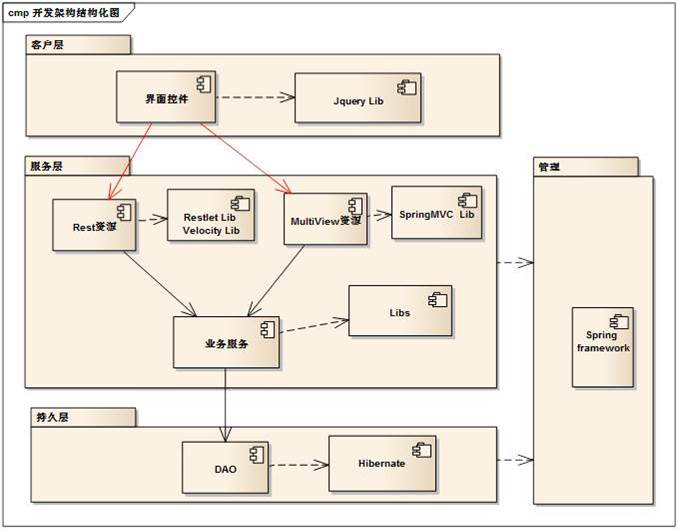
</beans>

其中key属性映射了对于的url模式，bean属性指定了对应的Restlet资源Bean名称。url模式中最后面的指示符{method}，说明此处的url字符映射了资源类的方法名称。例如，下面的URL

http://域名：端口/restful/privilegeProxyService/offic /index

* 请求，会映射到UserResource类上的loginUI (BaseRequest request, BaseResponse response)方法上处理。

## 开发架构机制描述



如上图所示，开发框架基于轻量级开源框架进行构造。

客户端基于开源的JQuery库为基础，建设客户端控件库。

服务器端，可以分为请求处理、业务服务、持久化三层次。其中，请求处理层，为了即支持高效的接口编程又支持丰富的人机UI实现，提供了两种实现机制。基于Restlet框架实现的请求处理层封装，适合于高效远程接口的模式，并且基于Velocity模板引擎实现UI界面；基于Spring mvc实现的请求处理层支持传统的JSP、Velocity、Freemarker等模板技术，并且提供页面流转机制，适合传统企业信息项目的建设。在编程模式、请求参数封装、上下文获取等方面两套请求处理机制，在开发模型上保持了一致性，开发人员继承不同的基类实现自己的请求处理类即可。请求层处理都依赖相同的业务服务类，业务服务对象依赖框架插件与第三方库实现自己的业务逻辑。持久化层的数据访问对象基于Hibernate框架实现OR映射与持久化功能，对上层提供功能支持。所以的服务器端对象的依赖性与生命周期基于Spring框架实现。

### 对象通用封装

在一次标准的用户请求的处理过程中，涉及到四个层次的处理：客户层（浏览器）、视图层（请求处理）、业务逻辑层（业务处理），数据访问层（持久化访问）。在不同层次间传递参数，会涉及到多次类型转换。在面向对象的实现中，为了增加接口的通用性会推荐以对象的方式传递参数，这时会带来更多类型转换与处理的负担。

一半以上的业务操作都是以查询为主，从数据库查询数据并显示到页面上，过多的类型转换只能带给开发者无谓的负担，值对象的通用封装就是为了解决这个问题。具体策略就是：数据访问层、业务服务层、表示层的值对象采用HashMap的数据结构封装，表示层面与客户层的显示值对象以Json对象的格式进行封装。由于Json对象实现了HashMap的接口，所以大部分使用场景下这两类对象可以互换，如果需要转换代价也很小，框架直接提供转换方法。

对象是属性与方法的结合，HashMap以Key-Value的方式可以表任何一个对象的数据属性集，对应到Key-Value方式的Json对象，在对应到客户端的Key-Value方式的Json字符串，其实都是统一数据结构的不同表示，对应的一致化的程序抽象，这样可以自动化参数转化过程，有降低开发负担。例如框架提供的分页支持，就是采用这种方式保存查询结果到Page对象中

/\*\*

\* 获取查询列表

\* **@param** paramMap

\* **@return**

\*/

**public** Page querys(Map<String, Object> params);

业务层查询结果会直接封装到视图层，并保存到视图解析上下文中

/\*\*

\* 获取学生列表信息

\* **@return**

\*/

@RequestMapping(value = "studentList")

@ResponseBody

**public** String studentInfo(**final** ModelMap modelMap){

Msg msg = **new** Msg();

msg.setSuccess(**true**);

Page page = studentService.querys(getJsonObject());

msg.setData(page);

**return** msg.toJSONObject("yyyy-MM-dd").toString();

}

在视图模板中可以直接从上下文提出字段值通过tmpl渲染到页面中

$("#listTmpl").tmpl(result.data.items).appendTo(tbody);

<!-- 列表模板 -->

<script type=*"text/x-jquery-tmpl"* id=*"listTmpl"*>

<tr class="active" id="${ID}">

<td>${ID}</td>

<td>${NAME}</td>

<td>${CLASSNAME}</td>

<td>{{if SEX=='1'}}男{{else}}女{{/if}}</td>

<td>${BIRTHDAY}</td>

<td>${DESCRIPTION}</td>

</tr>

</script>

在上面的场景中需要说明，字段值直接用字段名获取，为了适应不同的数据库引擎（Oracle、Mysql等），以及不同的部署系统（Windows、Linux等），框架在后台直接将字段都转换成大写字符。

框的提供的数据访问对象DAO的基类中提供了丰富的数据访问方法，基本满足了业务需求，开发人员不需要自己实现,只需继承HibernateEntityDao。这里要注意的是以Sql为前缀的方法返回通用值对象的数据封装，以Query为前缀的方法返回业务实体对象的数据封装。主要方法说明如下：

/\*\*

\* **@param** hql

\* HQL 语言

\* **@param** args

\* 传递参数

\* **@return** 返回更新记录数

\*/

**public** **int** update(String hql, Object[] args);

/\*\*

\* 根据 HQL 批删除记录

\* **@param** hql

\* HQL 语言

\* **@param** args

\* 传递参数

\* **@return** 返回批删除记录数

\*/

**public** **int** delete(String hql, Object[] args) ;

/\*\*

\* 返回单实体类hql

\* **@param** select 查询语句

\* **@param** values 参数

\* **@return**

\*/

**public** Object findForObject(String select, Object[] values) ;

/\*\*

\* 获取分页查询

\* **@param** hql 查询 语句

\* **@return**

\*/

**public** Page queryForpage(String hql);

/\*\*

\* 获取分页查询

\* **@param** hql 查询 语句

\* **@param** orderby 排序

\* **@return**

\*/

**public** Page queryForpage(String hql, LinkedHashMap<String, String> orderby) ;

/\*\*

\* 获取分页查询

\* **@param** hql 查询 语句

\* **@param** params 参数

\* **@param** orderby 排序

\* **@return**

\*/

**public** Page queryForpage(String hql, Object[] params,

LinkedHashMap<String, String> orderby);

/\*\*

\* 获取分页查询

\* **@param** hql 查询 语句

\* **@param** offset 页码

\* **@param** pagesize 条数

\* **@param** orderby 排序

\* **@return**

\*/

**public** Page queryForpage(String hql, **int** offset, **int** pagesize,

LinkedHashMap<String, String> orderby) ;

/\*\*

\* 获取分页查询

\* **@param** hql 查询 语句

\* **@param** values 参数

\* **@param** offset 页码

\* **@param** pagesize 条数

\* **@param** orderby 排序

\* **@return**

\*/

**public** Page queryForpage(String hql, Object values, **int** offset,

**int** pagesize, LinkedHashMap<String, String> orderby);

}

/\*\*

\* 获取分页查询

\* **@param** select 查询语句

\* **@param** values 参数

\* **@param** pagestart 页码

\* **@param** pagesize 条数

\* **@param** orderby 排序

\* **@return**

\*/

**public** Page queryForpage(String select, Object[] values, **int** pagestart, **int** pagesize, LinkedHashMap<String, String> orderby);

/\*\*

\* 通过sql查询分页列表

\* **@param** listSql

\* **@param** countSql

\* **@param** rowMapper

\* **@param** orderby

\* **@return**

\*/

**public** Page sqlQueryForPage(String sql, Object[] params,

LinkedHashMap<String, String> orderby);

}

/\*\*

\* 通过sql查询分页列表

\* **@param** listSql

\* 获取列表的SQL

\* **@param** countSql

\* 查询总数的SQL

\* **@param** rowMapper

\* 结果数组转化成对象

\* **@param** pagestart

\* **@param** pagesize

\* **@param** orderby

\* **@return**

\*/

**public** Page sqlQueryForPage(String sql, Object[] params, **int** offSet,

**int** pagesize, LinkedHashMap<String, String> orderby) ;

/\*\*

\* 返回实体类sql

\* **@param** select 查询语句

\* **@param** values 参数

\* **@return**

\*/

**public** Object sqlFindObject(String select, Object[] values) ;

/\*\*

\* 获取查询列表sql

\* **@param** sql 查询语句

\* **@param** params 参数

\* **@param** orderby 排序

\* **@return**

\*/

**public** List<Map<String, Object>> sqlQueryForList(String sql,

Object[] params, LinkedHashMap<String, String> orderby) ;

}

/\*\*

\* 获取查询列表sql

\* **@param** sql 查询语句

\* **@param** params 参数

\* **@param** offSet 页码

\* **@param** pagesize 显示条数

\* **@param** orderby 排序

\* **@return**

\*/

**public** List<Map<String, Object>> sqlQueryForList(String sql,

Object[] params, Integer offSet, Integer pagesize,

LinkedHashMap<String, String> orderby) ;

/\*\*

\* 查询分页列表hql

\* **@param** criteriaModel

\* **@return**

\* **@throws** Exception

\*/

**public** List queryList(CriteriaModel criteriaModel)**throws** Exception{

**return** **this**.queryList(criteriaModel, getReferenceClass());

}

/\*\*

\* 查询分页hql

\* **@param** criteriaModel 查询实体类

\* **@return**

\* **@throws** Exception

\*/

**public** Page queryPage(CriteriaModel criteriaModel);

/\*\*

\* 查询分页hql

\* **@param** criteriaModel 查询实体类

\* **@param** classes 返回实体对象

\* **@return**

\* **@throws** Exception

\*/

**public** Page queryPage(CriteriaModel criteriaModel, Class classes)**;**

/\*\*

\* 分页查询sql

\* **@param** criteriaModel 查询的实体类

\* **@return**

\* **@throws** Exception

\*/

**public** Page sqlQueryPage(CriteriaModel criteriaModel);

/\*\*

\* 分页查询sql

\* **@param** criteriaModel 查询实体类

\* **@param** classes 返回的实体对象

\* **@return**

\* **@throws** Exception

\*/

**public** Page sqlQueryPage(CriteriaModel criteriaModel,Class classes) **;**

### 请求参数数据绑定

针对get和post的提交方式，框架对接受的参数进行了一致性的封装，服务器端通过getJsonObject()而获取所有的参数。

开发人员实现Restlet资源时，需要继承框架BaseResource基类，实现controller资源时，需要继承框架MultiViewResource基类；

开发人员直接调用方法getJsonObject()获得参数的方式如下：

1、获取前端传递的所有参数：

Page page = studentService.querys(getJsonObject());

2、参数转换为类对象，通过JSONTools工具直接转换。

Student student = JSONTools.JSONToBean(getJsonObject(), Student.class);

3、获得某个参数值

String id = getJsonObject().getString("id")

4、附件上传的情况，获得封装的FileAttach即可。

获取方法：List<FileAttach> fileList = getfiles();

通过IoUtil工具类将FileAttach中的流直接写入文件中。方法如下：

IoUtil.createFile(new File(descfile, newfilename), fileItem.getInputStream());

5、@RequestParam绑定单个请求参数值，用于将请求参数区数据映射到功能处理方法的参数上。例如：

public String requestparam(@RequestParam String username)

说明：请求中包含username参数（如/requestparam1?username=zhang），则自动传入。

public String requestparam(@RequestParam("username") String username)；

说明：即通过@RequestParam("username")明确告诉使用username进行入参。

public String requestparam(@RequestParam(value="username", required =false) String username);

原子类型：必须有值,否则抛出异常,如果允许空值请使用包装类代替。

Boolean包装类型类型默Boolean.FALSE,其他引用类型默认为null。

public String requestparam(@RequestParam(value="role") String[] roleList);

说明：请求参数类似于url?role=admin&rule=user，则实际roleList参数入参的数据为“admin,user”，即多个数据之间使用“，”分割;

6、@CookieValue绑定Cookie数据值，用于将请求的Cookie数据映射到功能处理方法的参数上。

public String test(@CookieValue(value="JSESSIONID", defaultValue="") String sessionId)

说明：将自动将JSESSIONID值入参到sessionId参数上defaultValue

表示Cookie中没有JSESSIONID时默认为空。

public String test(@CookieValue(value="JSESSIONID", defaultValue="") Cookie sessionId)

说明：传入参数类型也可以是javax.servlet.http.Cookie类型。

7、@RequestHeader绑定请求头数据，用于将请求的头信息区数据映射到功能处理方法的参数上。

@RequestMapping(value="/header")

public String test(@RequestHeader("User-Agent") String userAgent,

@RequestHeader(value="Accept") String[] accepts) ;

8、@ModelAttribute绑定请求参数到命令对象

具有如下三个作用：

①绑定请求参数到命令对象：放在功能处理方法的入参上时，用于将多个请求参数绑定到一个命令对象，从而简化绑定流程，而且自动暴露为模型数据用于视图页面展示时使用；

②暴露表单引用对象为模型数据：放在处理器的一般方法（非功能处理方法）上时，是为表单准备要展示的表单引用对象，如注册时需要选择的所在城市等，而且在执行功能处理方法（@RequestMapping注解的方法）之前，自动添加到模型对象中，用于视图页面展示时使用；

③暴露@RequestMapping方法返回值为模型数据：放在功能处理方法的返回值上时，是暴露功能处理方法的返回值为模型数据，用于视图页面展示时使用。

1. 绑定请求参数到命令对象

如用户登录，我们需要捕获用户登录的请求参数（用户名、密码）并封装为用户对象，此时我们可以使用@ModelAttribute绑定多个请求参数到我们的命令对象。例如：

public String test(@ModelAttribute("user") User user)

说明:将该绑定的命令对象以“user”为名称添加到模型对象中供视图页面展示使用。我们此时可以在视图页面使${user.username}来获取绑定的命令对象的属性。

1. 暴露表单引用对象为模型数据

@ModelAttribute("user") //①

public UserModel getUser(@RequestParam(value="username", defaultValue="") String username) {

//TODO 去数据库根据用户名查找用户对象

UserModel user = new UserModel();

user.setRealname("zhang");

return user; }

说明：如你要修改用户资料时一般需要根据用户的编号/用户名查找用户来进行编辑，此时可以通过如上代码查找要编辑的用户。也可以进行一些默认值的处理。

1. 暴露@RequestMapping方法返回值为模型数据

public @ModelAttribute("user2") User test(@ModelAttribute("user2") User user)；

说明：返回值类型是命令对象类型，而且通@ModelAttribute("user2")注解，此时会暴露返回值到模型数据（名字为user2）中供视图展示使用。

由上面描述可见，框架对不同请求场景下的参数进行封装，为开发人员提供灵活的请求参数的方式，简化了编程。

### 注解式控制器的数据验证、类型转换及格式化

在Web/客户端项目中,通常需要将数据转换为具有某种格式的字符串进行展示，为了满足开发者，完成这个需求。

#### 数据类型转换及格式化

###### **类型级别的解析/格式化**

//CurrencyFormatter：实现货币样式的格式化/解析

CurrencyFormatter currencyFormatter = new CurrencyFormatter();

currencyFormatter.setFractionDigits(2);//保留小数点后几位

currencyFormatter.setRoundingMode(RoundingMode.CEILING);//舍入模式（ceilling表示四舍五入）

1、将带货币符号的字符串“$123.125”转换为BigDecimal("123.00")

Assert.assertEquals(new BigDecimal("123.13"), currencyFormatter.parse("$123.125", Locale.US));

2、将BigDecimal("123")格式化为字符串“$123.00”展示

Assert.assertEquals("$123.00", currencyFormatter.print(new BigDecimal("123"), Locale.US));

Assert.assertEquals("￥123.00", currencyFormatter.print(new BigDecimal("123"), Locale.CHINA));

Assert.assertEquals("￥123.00", currencyFormatter.print(new BigDecimal("123"), Locale.JAPAN));

说明：

parse方法：将带格式的字符串根据Locale信息解析为相应的BigDecimal类型数据；

print方法：将BigDecimal类型数据根据Locale信息格式化为字符串数据进行展示。

###### **字段级别的解析/格式化**

(1)使用内置的注解进行字段级别的解析/格式化：

public class FormatterModel {

@NumberFormat(style=Style.NUMBER, pattern="#,###")

private int totalCount;

@NumberFormat(style=Style.PERCENT)

private double discount;

@NumberFormat(style=Style.CURRENCY)

private double sumMoney;

@DateTimeFormat(iso=ISO.DATE)

private Date registerDate;

@DateTimeFormat(pattern="yyyy-MM-dd HH:mm:ss")

private Date orderDate;

//省略getter/setter }

说明：

@Number：定义数字相关的解析/格式化元数据（通用样式、货币样式、百分数样式），参数如下：

style：用于指定样式类型，包括三种：Style.NUMBER（通用样式） St yle.CURRENCY（货币样式） Style.PERCENT（百分数样式），默认Style.NUMBER；

pattern：自定义样式，如patter="#,###"；

@DateTimeFormat：定义日期相关的解析/格式化元数据，参数如下：

pattern：指定解析/格式化字段数据的模式，如”yyyy-MM-dd HH:mm:ss”；

iso：指定解析/格式化字段数据的ISO模式，包括四种：ISO.NONE（不使用）ISO.DATE(yyyy-MM-dd) ISO.TIME(hh:mm:ss.SSSZ) ISO.DATE\_TIME(yyyy-MM-dd hh:mm:ss.SSSZ)，默认ISO.NONE；

style：指定用于格式化的样式模式，默认“SS”，具体使用请参考Joda-Time类库的org.joda.time.format.DateTimeFormat的forStyle的javadoc；

优先级： pattern 大于 iso 大于 style。

###### **自定义注解进行字段级别的解析/格式化：**

、定义解析/格式化字段的注解类型：

@Target({ElementType.METHOD, ElementType.FIELD, ElementType.PARAMETER})

@Retention(RetentionPolicy.RUNTIME)

public @interface PhoneNumber { }

、实现AnnotationFormatterFactory注解格式化工厂：

public class PhoneNumberFormatAnnotationFormatterFactory

implements AnnotationFormatterFactory<PhoneNumber> {

private final Set<Class<?>> fieldTypes;

private final PhoneNumberFormatter formatter;

public PhoneNumberFormatAnnotationFormatterFactory() {

Set<Class<?>> set = new HashSet<Class<?>>();

set.add(PhoneNumberModel.class);

this.fieldTypes = set;

//此处使用之前定义的Formatter实现

this.formatter = new PhoneNumberFormatter();

}

//指定可以被解析/格式化的字段类型集合

@Override

public Set<Class<?>> getFieldTypes() {

return fieldTypes;

}

//根据注解信息和字段类型获取解析器

@Override

public Parser<?> getParser(PhoneNumber annotation, Class<?> fieldType) {

return formatter;

}

//根据注解信息和字段类型获取格式化器

@Override

public Printer<?> getPrinter(PhoneNumber annotation, Class<?> fieldType) {

return formatter;

}

}

#### 数据验证

##### **编程式数据验证**

@NotNull(message = "用户名不能为空")

@Length(min=5, max=20, message="用户名长度必须在5-20之间")

@Pattern(regexp = "^[a-zA-Z\_]\\w{4,19}$", message = "用户名必须以字母下划线开头，可由字母数字下划线组成")

private String username;

如上所示，错误消息使用硬编码指定，这种方式是不推荐使用的，因为在如下场景是不适用的：

1、在国际化场景下，需要对不同的国家显示不同的错误消息；

2、需要更换错误消息时是比较麻烦的，需要找到相应的类进行更换，并重新编译发布。

解决方式：从资源消息文件中根据消息键读取错误消息

（1、定义错误消息文件ValidationMessages.properties

在类装载路径的根下创建ValidationMessages.properties文件，如在src目录下创建会自动复制到类装载路径的根下，并添加如下消息键值（需要native2ascii，可以在eclipse里装Properties Editor，自动保存为ASCII码）

javax.validation.constraints.Pattern.message=用户名必须以字母或下划线开头，后边可以跟字母数字下划线，长度在5-20之间

需要在你spring配置文件sprin-servlet.xml修改之前的validator Bean：

<bean id="validator"class="org.springframework.validation.

beanvalidation.LocalValidatorFactoryBean">

<property name="providerClass" value=

"org.hibernate.validator.HibernateValidator"/>

</bean>

此时错误消息键值的查找会先到classpath下ValidationMessages.properties中找，找不到再到默认的错误消息文件中找。

（2、使用Spring的MessageSource Bean进行消息键值的查找

如果我们的环境是与spring集成，还是应该使用Spring提供的消息支持，具体配置如下：

在spring配置文件spring-servlet.xml定义MessageSource Bean：

<bean id="messageSource" class="org.springframework.context

.support.ReloadableResourceBundleMessageSource">

<property name="basename" value="classpath:messages"/>

<property name="fileEncodings" value="utf-8"/>

<property name="cacheSeconds" value="120"/>

</bean>

（3、自定义错误消息键值

硬编码错误消息，及默认的错误消息，在大部分场景下，以上两种方式无法满足我们的需求，因此我们需要自定义错误消息键值。在验证约束注解上指定错误消息键。

public class PatternModel {@Pattern(regexp= "^[a-zA-Z\_][\\w]{4,19}$", message="{user.name.error}")

private String value;

}

##### **声明式数据验证**

（1、添加jar包：

使用Hibernate-validator实现，将如下jar包添加到classpath（WEB-INF/lib下即可）：

（2、在Spring配置总添加对JSR-303验证框架的支持

<!-- 以下validatorConversionService 在mvc:annotation-driven会自动注册-->

<bean id="validator" class="org.springframework.validation.beanvalidation.

LocalValidatorFactoryBean">

<property name="providerClass" value="org.hibernate.validator.HibernateValidator"/>

<!-- 如果不加默认到 使用classpath下的 ValidationMessages.properties -->

<property name="validationMessageSource" ref="messageSource"/>

</bean>

注意：此处使用Hibernate validator实现：

validationMessageSource属性：指定国际化错误消息从哪里取，此处使用之前定义的messageSource来获取国际化消息；如果此处不指定该属性，则默认到classpath下的ValidationMessages.properties取国际化错误消息

通过ConfigurableWebBindingInitializer注册validator：

<bean id="webBindingInitializer"

class="org.springframework.web.bind.support.

ConfigurableWebBindingInitializer">

<property name="conversionService" ref="conversionService"/>

<property name="validator" ref="validator"/>

</bean>

（3、使用JSR-303验证框架注解为模型对象指定验证信息

public class UserModel {

@NotNull(message="{username.not.empty}")

private String username;

}

说明：通过@NotNull指定此username字段不允许为空，当验证失败时将从之前指定的messageSource中获取“username.not.empty”对于的错误信息，此处只有通过“{错误消息键值}”格式指定的才能从messageSource获取。

（4、控制器

@Controller

public class HelloWorldController {

@RequestMapping("/validate/hello")

public String validate(@Valid @ModelAttribute("user") UserModel user, Errors errors) {

if(errors.hasErrors()) {

return "validate/error";

}

return "redirect:/success";

}

}

通过在命令对象上注解@Valid来告诉controller此命令对象在绑定完毕后需要进行JSR-303验证，如果验证失败会将错误信息添加到errors错误对象中。

（5、验证失败后需要展示的页面（/WEB-INF/validate/error.）

<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>

<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

<form:form commandName="user">

<form:errors path="\*" cssStyle="color:red"></form:errors><br/>

</form:form>

### 服务消息对象封装

针对服务接口编程，规范对外提供服务格式的标准的一致性消息类对象。

服务消息格式：{success:true,code:XXX，msg:xx,data:{ } }

* 属性说明

success：表示接口请求的成功（true）与失败(false)状态

code：可以对接口请求的响应状态进行细节编码例如：http协议的402、403等状态编码，用户也可根据自定的情况定义业务编码。

msg：是对返回消息的描述，友好提示的简述。

data：具体的消息本身的内容。

* 方法说明

/\*\*

\* 将消息转为JsonObject对象。

\* 日期格式化

\* @return JSONObject

\*/

public JSONObject toJSONObject()；

/\*\*

\* 将消息转为JsonObject对象。

\* 自定义日期格式化

\* @return JSONObject

\*/

public JSONObject toJSONObject(String dataFormat) ；

基于这个消息对象进行接口编程可以提供开发人员的效率与规范性，增强接口实现的稳定性。应用实例如下：

Msg msg = new Msg();

msg.setSuccess(true);

msg.setMsg("保存成功！");

msg.setData(student);

msg.toJSONObject();

上例中，客户端浏览器可以接受到提示文本为“保存成功”的成功响应消息。

/\*\*

\* 将消息转换为JSON

\* Presention格式:{success:true,msg:xx,data:{xx1:value1,....,xx:value}}

\* **@param** names

\* 转换为json对象的属性列表

\* **@return**

\*/

**public** Representation toJSONObjectPresention() ;

/\*\*

\* 将消息转换为JSON

\* Presention格式:{success:true,msg:xx,data:{xx1:value1,xx:value}}

\* **@param** names

\* 转换为json对象的属性列表

\* **@return**

\*/

**public** Representation toJSONObjectPresention(String dataFormat);

### 快速查询通道

在开发过程中，经常遇到一种情况就是，一次请求就是执行一个查询，并且对事务的要求只读型事务。这种情况下，按照架构规范定义Service层、Dao层类去实现，是得不偿失的。对此，框架定义了一种快速查询的模式：cn.newcapec.framework.core.dao.db. SqlDataset

/\*\*

\* 构造方法.

\* **@param** sql

\* 查询sql

\* **@param** params

\* 查询参数.

\* **@param** supportPaging

\* 是否需要分页，默认为true.

\* **@param** start

\*请求的开始记录条数,默认为0.

\* **@param** pageSize

\* 每页记录数，默认为20条.

\*/

**public** SqlDataset(String sql, ParameterSet params, **boolean** supportPaging,**int** start, **int** pageSize) ;

/\*\*

\* 构造方法.

\* **@param** sql

\* 查询sql

\* **@param** params

\* 查询参数.

\* **@param** supportPaging

\* 是否需要分页.

\* **@param** start

\* 请求的开始记录条数,默认为0.

\* **@param** pageSize

\* 每页记录数，默认为20.

\*/

**public** SqlDataset(String sql, ParameterSet params, **boolean** supportPaging, String start, String pageSize) ;

/\*\*

\* 构造方法.

\* **@param** sql

\* 查询sql

\* **@param** params

\* 查询参数.

\* **@param** start

\* 请求的开始记录条数,默认为0.

\*/

**public** SqlDataset(String sql, ParameterSet params) ;

/\*\*

\* 构造方法.

\* **@param** sql

\* 查询sql

\*/

**public** SqlDataset(String sql) ;

提供的常用的方法：

/\*\*

\* 加载数据

\*/

**public** DbDataset loadData() ;

/\*\*

\* 转换为分页

\* **@return**

\*/

**public** Page toPage();

/\*\*

\* 设置sql

\* **@param** sql

\*/

**public** DbDataset setSql(String sql);

/\*\*

\* 返回结果集数据

\*

\* **@return**

\*/

**public** List getData();

/\*\*

\* 将查询结果转换为JSONObject,格式:{total:xx,data:[{name:value,name1:value1,...},..,{}

\* ]}

\* 转换为json对象的属性列表

\* **@return**

\*/

**public** JSONObject toJSONObject(String[] names) ;

/\*\*

\* 将查询结果转换为JSONArray数组，格式[{},{}]

\* **@param** names

\* 转换为json对象的属性列表

\* **@return**

\*/

**public** JSONArray toJSONArray(String[] names);

/\*\*

\* 设置数据结果集DO对象类型

\* **@param** clazz

\* **@return**

\*/

**public** DbDataset setClazz(Class clazz);

开发人员从视图层获得SqlDataset，根据业务需求确定返回的数据类型，而设置返回的list列表，呈现在界面中。

### 异常处理

系统异常.这是一个运行时异常，代表系统原因产生的异常，比如SQLException产生的异常，网络访问失败产生的异常，等。这种异常，程序中无法编程处理，用户也无法恢复，因此作为运行时异常抛出.

**public** **class** SysException **extends** RuntimeException {

/\*\*

\* 构造函数.

\*/

**public** SysException() {

**super**();

}

/\*\*

\* 构造函数.

\* **@param** msg

\*异常信息.

\*/

**public** SysException(String msg) ;

/\*\*

\* 构造函数.

\* **@param** cause

\* 产生本异常的根异常实例.

\*/

**public** SysException(Throwable cause) ;

/\*\*

\* 构造函数.

\* **@param** msg

\*异常信息.

\* **@param** cause

\*产生本异常的根异常实例.

\*/

**public** SysException(String msg, Throwable cause);

/\*\*

\* 设置异常消息的keyCode，主要为支持国际化.

\* **@param** key

\* 异常消息的keyCode.

\*/

**public** **void** setMessageKey(String key) ;

/\*\*

\* 获得异常的keyCode，主要为了支持国际化.

\* **@return** 异常消息的keyCode.

\*/

**public** String getMessageKey();

/\*\*

\* 设置异常消息的参数，主要为了支持国际化.

异常消息使用如下格式定义带参数的消息: {0} is larger than {1}.

\* **@param** args

\* 异常消息的参数数组.

\*/

**public** **void** setMessageArgs(Object[] args) ;

/\*\*

\* 获得异常消息的参数，主要为了支持国际化.

\* **@return** 异常消息的参数数组.

\*/

**public** Object[] getMessageArgs() ;

/\*\*

\* 设置根异常.

\* **@param** anException

\*根异常.

\*/

**public** **void** setRootCause(Throwable anException);

/\*\*

\* 获得根异常.

\* **@return** 根异常.

\*/

**public** Throwable getRootCause();

/\*\*

\* 将异常的栈信息打印到System.err，

先打印本异常的栈信息，然后打印根异常的栈信息.

\*/

**public** **void** printStackTrace();

/\*\*

\* 将异常的栈信息打印到输出流，先打印本异常的栈信息，然后打印根异常的栈信息.

\* **@param** outStream

\* 输出流.

\*/

**public** **void** printStackTrace(PrintStream outStream) ;

/\*\*

\* 将异常的栈信息打印到输出流，先打印本异常的栈信息，然后打印根异常的栈信息.

\* **@param** writer

\* 输出流

\*/

**public** **void** printStackTrace(PrintWriter writer);

}

针对RestLet资源处理，在config/restlet-conf/restletContext.xml中配置了框架的异常处理服务，将捕获所有的异常，返回友好提示给客户端

<bean id=*"exceptionStatusService"* class=*"cn.newcapec.framework.core.exception.ExceptionStatusService" /*>

针对MultiView资源处理，在

config/spring-conf/applicationContext.xml中进行配置，让Spring mvc不好框架定义异常，一同进行处理

<!-- 异常类的处理handle -->

<bean class=*"cn.newcapec.framework.core.handler.SystemExceptionHandler"* />

### 系统异常断言

异常抛出、异常捕获、友好性提示转换在编程过程，是一个代码重重复率很高的过程。为了简化这部分的工作，框架提供了异常断言类，配合框架基类中的异常断言处理方法来解决这个问题

在资源基类中都实现了用以断言处理方法能doExpAssert（），其只有一个传入参数为实现AssertObject接口的对象。

AssertObject接口中定义了一个断言方法AssertMethod（Msg paramMsg），在该方法中实现需要异常处理的业务逻辑，系统可以自动完成异常捕获与处理逻辑。使用时，可以使用内嵌匿名类的简化编程方式，如下

/\*\*

\* 更新学生信息

\* **@param** request

\* **@param** response

\* **@return**

\*/

@RequestMapping(value = "edit",method=RequestMethod.*POST*)

@ResponseBody

**public** Msg edit(**final** Student student) {

**return** doExpAssert(**new** AssertObject() {

@Override

**public** **void** AssertMethod(Msg Msg) {

studentService.saveOrUpdate(student);

}

});

}

在上面的例子中，Assert是系统提的另一个断言类，其中包含的Assert.isTrue(Boolean exp, Sting msg)方法，第一个参数是一个布尔表达式，当布尔表达式的值为false时保持提示为msg的异常。

### 视图地址统一管理：

框架无需再控制层硬编码来指定转发的目的页面，而是在属性文件中配置转发页面，这样不但提高了应用程序的灵活性，而且提高了应用程序的可维护性，编译日后的升级工作，在升级过程中我们只需要修改配置文件的信息就能是用户得到不同的显示页面，而不需要修改控制层（controller）的任何代码。

（1、创建地址属性文件：

在自己的业务资源文件下，根据业务名字前缀，建立URL属性配置文件即可。例如：portalUrlMapping.properties 内容格式如下：

#学生列表展示

student.stuListUI=foundation/student/pagelet/v1.0/stu\_list

#学生表格数据

student.stuListGrid=foundation/student/pagelet/v1.0/stu\_list\_grid

#学生添加页面

student.addStuUI=foundation/student/pagelet/v1.0/addStuUI

#学生编辑页面

student.editStuUI=foundation/student/pagelet/v1.0/editStuUI

（2、在Spring配置对属性文件内容的启动加载

<!-- 映射地址和视图展示 -->

<bean id="portalUrlMapping" class="cn.newcapec.framework.core.handler.ConfigurationRead">

<constructor-arg>

<list> <value>classpath:config/cn/newcapec/foundation/portal/controlle r-conf/portalUrlMapping.properties</value>

</list>

</constructor-arg>

</bean>

（3、用用程序调用方式：

/\*\*

\* 学生添加界面

\* @return String

\*/

@RequestMapping(value = "addStuUI")

public String addStuUI() {

return getUrl("student.addStuUI");

}

/\*\*

\* 更新学生界面

\* @return String

\*/

@RequestMapping(value = "editStuUI")

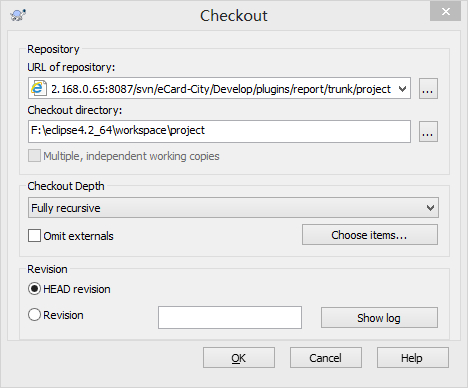
public String editStuUI() {

return getUrl("student.editStuUI");

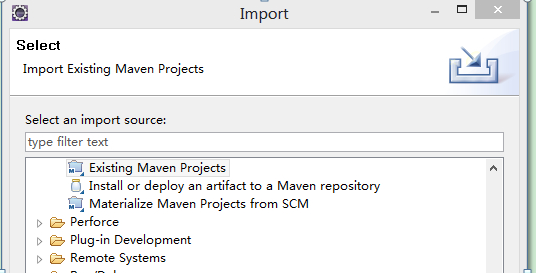
}

# 开发实例说明

（1、从svn库中Checkout模板工程



模板工程为maven项目，导入成eclipse中的maven工程



（2、数据源配置

在config/jdbc.properties中配置数据库链接属性、数据库连接池属性与Hibernate属性

######oracle数据库##################

biz.jdbc.driver.class.name=oracle.jdbc.driver.OracleDriver

biz.jdbc.url=jdbc:oracle:thin:@192.168.0.45:1521:NSP

biz.jdbc.username=NSP

biz.jdbc.password=root123

biz.hibernate.dialect=org.hibernate.dialect.Oracle10gDialect

######基本配置##################

c3p0.pool.maxsize=10

c3p0.pool.minsize=5

c3p0.pool.increment=5

c3p0.pool.inisize=3

######hibernate配置##################

biz.hibernate.show\_sql=true

biz.hibernate.cache.use\_query\_cache=false

biz.hibernate.cache.use\_second\_level\_cache=false

在config /spring-confi/dataAccessContext-hibernate.xml中业务实体。

<!--Hibernate SessionFatory 配置-->

<bean id="sessionFactory" class="cn.newcapec.framework.core.dao.hibernate.SessionFactoryAutoDialectBean" lazy-init="false">

<property name="dataSource" ref="bizDatasourceOracle"/>

<property name="packagesToScan">

指定到实体类的上级目录

<list>

<value>cn.newcapec.foundation.privilege.model</value>

<value>cn.newcapec.foundation.portal.model</value>

</list>

</property>

<property name="hibernateProperties">

<props>

<prop key="javax.persistence.validation.mode">none</prop>

<prop key="hibernate.show\_sql">true</prop>

<prop key="hibernate.hbm2ddl.auto">update</prop>

<prop key="hibernate.connection.release\_mode">after\_statement</prop>

</props>

</property>

</bean>

（3、定义业务实体

实体类继承CommonModel，主要实现了UUID的主键生成。

例如Student的实体模型：

数据库对应表

@Table(name = "t\_student")

@Entity

public class Student extends CommonModel {

private static final long serialVersionUID = 1L;

/\* 编号 \*/

@Column(name = "no", length = 30)

private String no;

/\* 班级 \*/

@Column(name = "className", length = 30)

private String className;

/\* 性别 \*/

@Column(name = "sex", length = 10)

private String sex;

/\* 名字 \*/

@Column(name = "name", length = 25)

private String name;

/\* 图片 \*/

@Column(name = "photo", length = 62)

private String photo;

/\* 描述 \*/

@Column(name = "description", length = 500)

private String description;

/\*\* 生日日期\*/

@JsonSerialize(using=CustomDateSerializer.class)

@Temporal(TemporalType.TIMESTAMP)

@Column(name="birthday")

private Date birthday;

//省略get/set……

}

注意：

1、AppBaseModel基类，实现了主键UUID生成方式，同时增加了创建者、创建时间、更新人、更新时间和版本字段。

2、根据主键的生成方式不同增加了CommonModelWithIntId，实现了自增长整形序列号的主键生成方式。

（4、实现业务数据访问DAO

为了增加系统的稳定性和重用性，抽取出HibernateEntityDao基类，作为公共DAO常用操作，开发人员继承此类即可。

/\*\*

\* 学生操作基础类

\*/

public abstract class BaseStudentDao extends HibernateEntityDao {

@Override

protected Class getReferenceClass() {

return cn.newcapec.foundation.portal.model.Student.class;

}

public cn.newcapec.foundation.portal.model.Student cast(Object object) {

return (cn.newcapec.foundation.portal.model.Student) object;

}

public cn.newcapec.foundation.portal.model.Student load(

java.io.Serializable key) {

return (cn.newcapec.foundation.portal.model.Student) load(

getReferenceClass(), key);

}

public cn.newcapec.foundation.portal.model.Student get(

java.io.Serializable key) {

return (cn.newcapec.foundation.portal.model.Student) get(

getReferenceClass(), key);

}

public java.util.List findAll() {

return find("from " + getReferenceClass().getName());

}

public void save(cn.newcapec.foundation.portal.model.Student student) {

super.save(student);

}

public void saveOrUpdate(cn.newcapec.foundation.portal.model.Student student) {

saveOrUpdate((Object) student);

}

public void update(cn.newcapec.foundation.portal.model.Student student) {

update((Object) student);

}

public void delete(java.io.Serializable id) {

delete((Object) load(id));

}

public void delete(cn.newcapec.foundation.portal.model.Student student) {

delete((Object) student);

}

这里有一个可选的设计，如果预测Dao类的实现以后会发生变化，出现同一个业务Dao接口根据业务场景不同，会出现不同的实现，可以定义一个体现稳定接口的抽象类BaseStudentDao

公共方法在基础业务Dao中实现，在具体类中专注实现业务方法就可以了，StudentDao实现如下：

/\*\*

\* 学生数据操作类

\* @author andy

\*/

@Repository

public class StudentDao extends BaseStudentDao {

/\*\*

\* 获取学生信息

\* @param paramMap

\* @return

\*/

public Page querys(Map<String, Object> paramMap) {

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

StringBuilder sql = new StringBuilder(

"select \* from t\_student t where 1=1");

if (null != paramMap) {

if (paramMap.containsKey("stuname")

&& StringUtils.isNotBlank(paramMap.get("stuname")

.toString())) {

sql.append(" and t.name=:stuname");

param.put("stuname", paramMap.get("stuname"));

}

}

return this.sqlqueryForpage(sql.toString(), param, null);

}

/\*\*

\* 批量删去学生信息

\* @param ids

\*/

public void delete(String[] ids) {

if (ids != null && ids.length > 0) {

Query query = getSession().createQuery(

"delete from Student s where s.id in (:ids)");

query.setParameterList("ids", ids);

query.executeUpdate();

}

}

}

注意：为了提高开发人员的开发效率，增加了一个DAO模板AppBaseDAO，实现了对实体类的基本（增加、删去、修改、查找）操作，但是这里 有一定的约束，实体类继承AppBaseModel。

（5、实现业务方法

实现业务服务对象StudentService，框架提供了一个BaseService接口

业务服务层实现，严格遵守接口与实现分离的原则，首先定义服务接口

/\*\*

\* 学生业务接口类

\* @author andy

\*

\*/

public interface StudentService extends BaseService<Student> {

/\*\*

\* 获取查询列表

\* @param paramMap

\* @return

\*/

public Page querys(Map<String, Object> paramMap);

/\*\*

\* 获取学生实体类

\* @param id

\* @return

\*/

public Student findById(String id);

/\*\*

\* 删去学生

\* @param ids

\*/

public void delete(String[] ids);

}

实现在具体的业务服务类中实现StudentServiceImpl

/\*\*

\* 学生业务接口实现类

\*/

@Service("studentService")

@Transactional

**public** **class** StudentServiceImpl **implements** StudentService {

Dao注入方式

@Autowired

**private** StudentDao studentDao;

@Override

**public** **void** removeUnused(String id) {

studentDao.delete(id);

}

@Override

@Transactional(readOnly=**true**,propagation = Propagation.*NOT\_SUPPORTED*)

**public** Student get(String id) {

**return** studentDao.get(id);

}

@Override

**public** **void** saveOrUpdate(Student paramT) {

studentDao.saveOrUpdate(paramT);

}

@Override

@Transactional(readOnly=**true**,propagation = Propagation.*NOT\_SUPPORTED*)

**public** Page querys(Map<String, Object> paramMap) {

Page page=studentDao.querys(paramMap);

**return** page;

}

@Override

@Transactional(readOnly=**true**,propagation = Propagation.*NOT\_SUPPORTED*)

**public** Student findById(String id) {

**return** studentDao.get(id);

}

@Override

**public** **void** delete(String[] ids) {

studentDao.delete(ids);

}

}

注意：系统的事务控制在service层，完全用注解（Transactional）的方式。而对于一些查询操作，不需要开启事务，需要在方法之上添加注解如下：

@Transactional(readOnly=**true**,propagation = Propagation.*NOT\_SUPPORTED*)即可。

（6、实现控制层

StudentController需要继承MultiViewResource基类。

/\*\*

\* 学生视图展示类

\* @author Administrator

\*

访问上下文

\*/

@Controller

@Scope("prototype")

@RequestMapping(value = "/student")

public class StudentController extends MultiViewResource {

业务类注入方式

@Autowired

private StudentService studentService;

/\*\*

\* 获取学生信息页面

\* @return

\*/

@RequestMapping(value = "studentInfo")

public String studentInfoUI(){

return this.toView("foundation/student/pagelet/v1.0/studentinfo");

}

/\*\*

\* 获取学生列表信息

\* @return

\*/

@RequestMapping(value = "studentList")

@ResponseBody

public String studentInfo(final ModelMap modelMap){

PageContext.setPagesize(JSONTools.getInt(getJsonObject(), "pageSize"));

Msg msg = new Msg();

msg.setSuccess(true);

Page page = studentService.querys(getJsonObject());

msg.setData(page);

return msg.toJSONObject("yyyy-MM-dd").toString();

}

/\*\*

\* 学生列表信息

\* @param request

\* @param response

\* @return

\*/

@RequestMapping(value = "stuListUI")

public ModelAndView stuListUI(ModelMap modelMap) {

Page page = studentService.querys(getJsonObject());

// 菜单列表视图

PageView<Map<String, Object>> pageView = new PageView<Map<String, Object>>(

PageContext.getPagesize(), PageContext.getOffset());

pageView.setQueryResult(page);

pageView.setJsMethod("reloadStuList");

modelMap.put("pageView", pageView);

return toView(getUrl("student.stuListUI"), modelMap);

}

}

这样，访问StudentController的studentInfo方法对应的URL为

http://域名：端口名/controller/student/ studentInfo

（7、视图展示层

列表的展示，本例子中，用到是jquery.tmpl,页面视图主要定义好模板即可：参考如下：

在视图模板中可以直接从上下文提出字段值通过tmpl渲染到页面中

$("#listTmpl").tmpl(result.data.items).appendTo(tbody);

<!-- 列表模板 -->

<script type=*"text/x-jquery-tmpl"* id=*"listTmpl"*>

<tr class="active" id="${ID}">

<td>${ID}</td>

<td>${NAME}</td>

<td>${CLASSNAME}</td>

<td>{{if SEX=='1'}}男{{else}}女{{/if}}</td>

<td>${BIRTHDAY}</td>

<td>${DESCRIPTION}</td>

</tr>

</script>

本例前端用到的是号称简洁、直观、强悍的前端开发框架，让web开发更迅速、简单的Bootstrap；

界面验证插件用到的是bootstrap3-validation.js 例如非空判断用法如下：

<input type="text" class="form-control" name="description" check-type="required" required-message="请填写描述"/>

**最后建议：人机交互，用controller视图开发。而系统之间的接口用restlet开发。**