文档编号：框架使用说明文档

文档类别：□公司级 □部门级 ■项目级 □普通级

保密级别：□绝密 □机密 □秘密 ■普通

框架使用说明文档

版本:4.0

2018-4-16

目录

1 框架整体介绍

框架基于 JDK8 平台,使用 Spring/SpringBoot 技术栈,Shiro 权限控制,Lucene 全文检索,Freemarker 模板等软件包实现,ORM 基于 Spring JDBC 封装实现,不使用 Hibernate 和 Ibatis/Mybatis.

1.1 基础软件包环境

软件名称	版本	用途	备注
JDK	1.8.31	Java 运行平台	
Tomcat	8.5.16	应用服务器	
MySQL	5.7.18	数据库	
Redis	4.0.1	缓存和消息服务器	非必选
Maven	3.5.0	项目构建	
SVN		版本控制	

1.2 项目依赖软件包

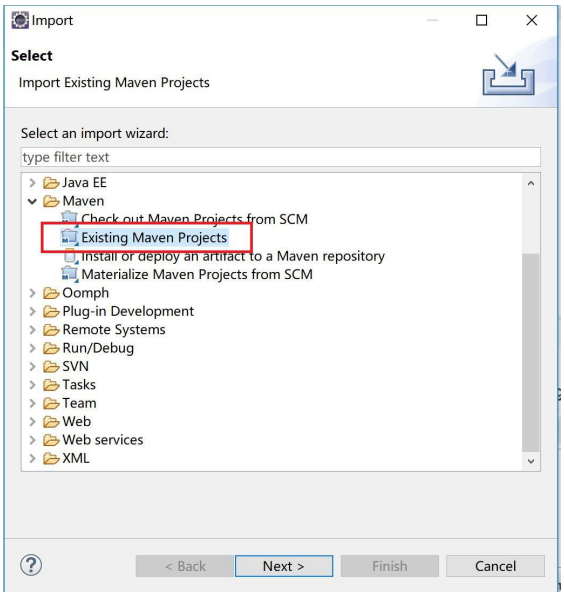
基于 freemarker 自定义标签,实现前后端开发隔离.

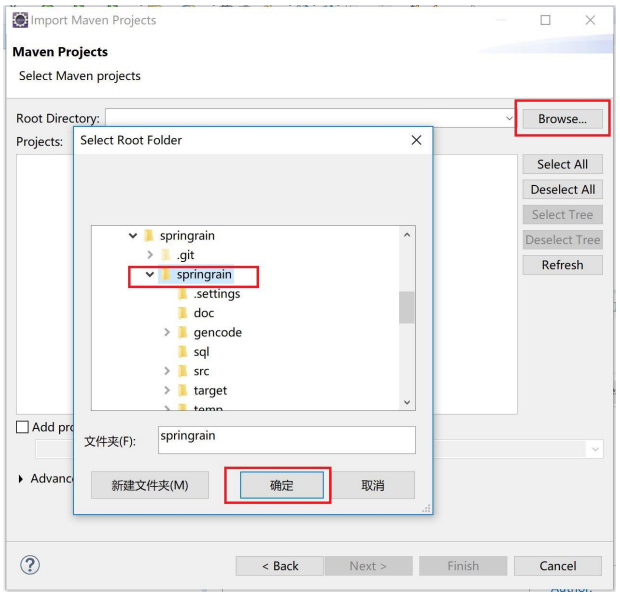
软件名称	版本	用途	备注
SpringBoot	2.0.1	Core,MVC,JDBC	
Quartz	2.2.3	调度框架	
Shiro	1.3.2	权限框架	

Freemarker	2.3.28	模板引擎	
Lucene	7.2.0	全文检索框架	
WeixinSDK	框架定制		
jQuery	1.11.2	JS 类库	
Weui	1.0.1	微信前端 UI 库	
UEditor	UEditor 定制		
LayUI	1.0.9-rls	后台 UI 库	

2 导入启动

Springrain 是一个完整的 eclipse 项目,直接使用 Eclipse 导入即可



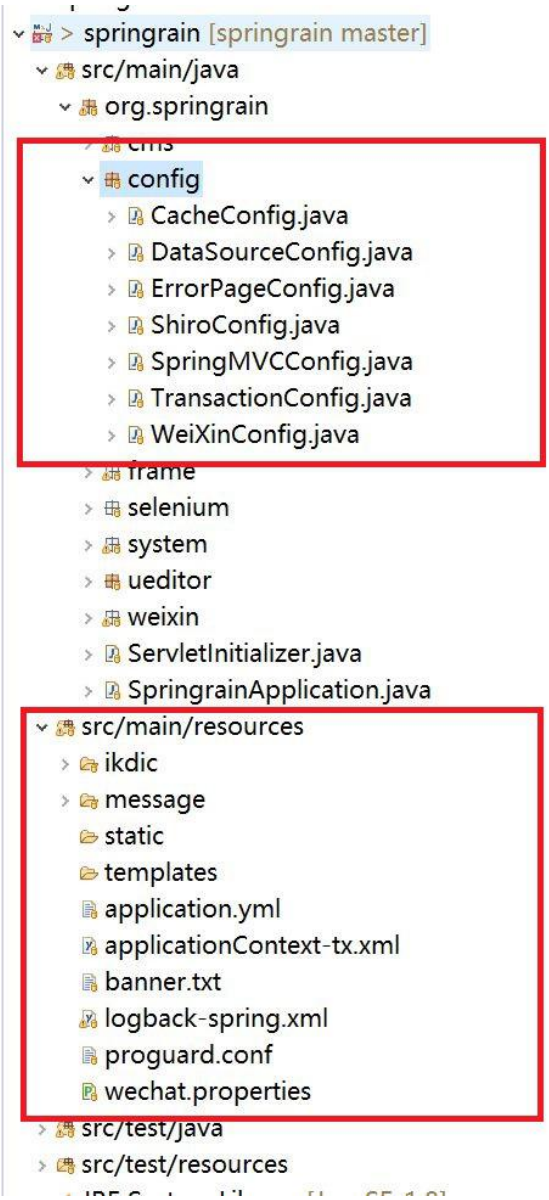


导入成功之后,运行 org.springrain.SpringrainApplication 类,就可以启动 springrain,默认端口是 8080,项目名是/,请根据实际情况修改

```java
package org.springrain;

import org.springframework.boot.SpringApplication;
/**
 * 主入口,排除@Controller注解,主要为了Controller指定命名规则.
 * 这个类所在的包,就是默认扫描的根包.
 * @author caomei
 *
 */
@SpringBootApplication
@ComponentScan(excludeFilters= {@Filter(type=FilterType.ANNOTATION,value=Controller.class)})
public class SpringrainApplication  {

    public static void main(String[] args) {
        SpringApplication.run(SpringrainApplication.class, args);
    }


}
```

3 主要配置

依据 Maven 目录结构,配置文件都在 src/main/resources 下,JavaConfig 配置类在 org.springrain.config 包下,主配置是 application.yml



3.1 ikdic

Lucene 中 IK 分词器的字典配置

main.dic 是主字典文件,quantifier.dic 是量词字典文件,stopword.dic 停止词字典文件.

3.2 message

用于国际化的资源文件,目前只有中文和英文.功能也不完善.

3.3 配置文件

application.yml SpringBoot 主配置

banner.txt 启动 banner 内容

logback-spring.xml MVC 的配置文件

proguard.conf 混淆配置文件.

wechat.properties 微信的配置文件.特殊情况可以配置微信服务的域名.

3.4 Java 配置类

Spring 的核心配置

CacheConfig.java 缓存配置

DataSourceConfig.java 数据库配置

ShiroConfig.java shiro 的权限配置

SpringMVCConfig.java MVC 的配置

TransactionConfig.java 事务配置,配合 applicationContext-tx.xml

WeiXinConfig.java 微信的配置

4 代码生成器（gencode）介绍

在项目的根目录下有 gencode 文件夹,generator.xml 是配置文件,rapid-gen.bat 是执行生成代码的命令文件

4.1 原理

.java .html .css .js 本质上都是文本文件,可以通过 freemarker 模板文件,生成对应的 java,html,css 和 js 文件.

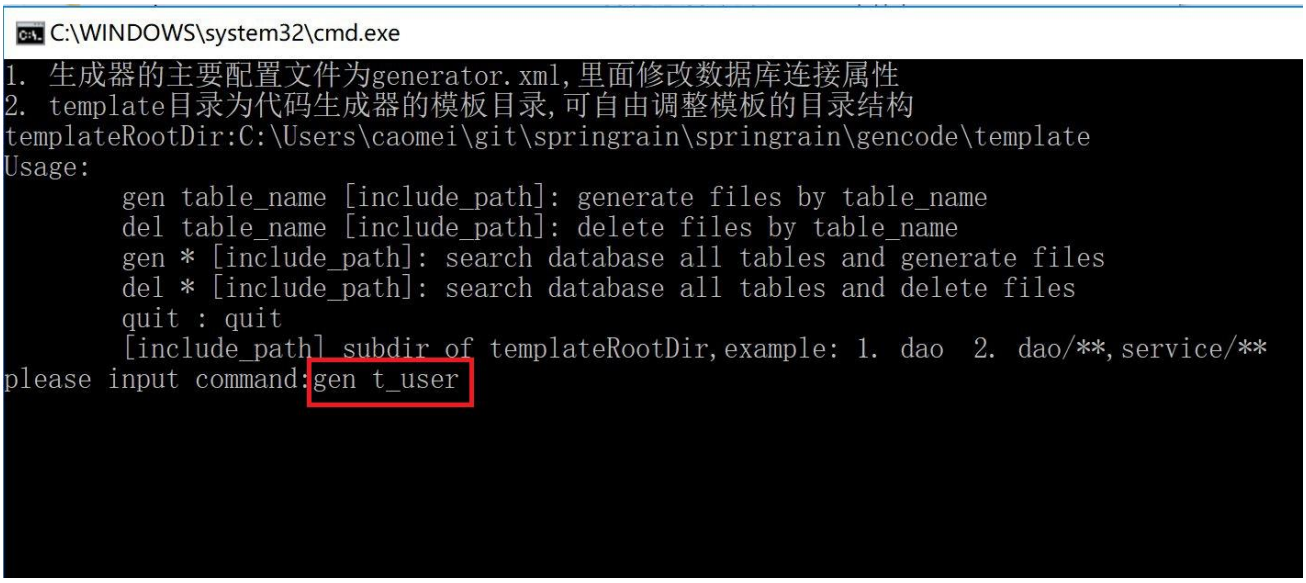
通过数据库连接,获取数据库表的元数据,例如:表的字段名称,中文注释等,依此生成对应的增删改查功能.

4.2 使用

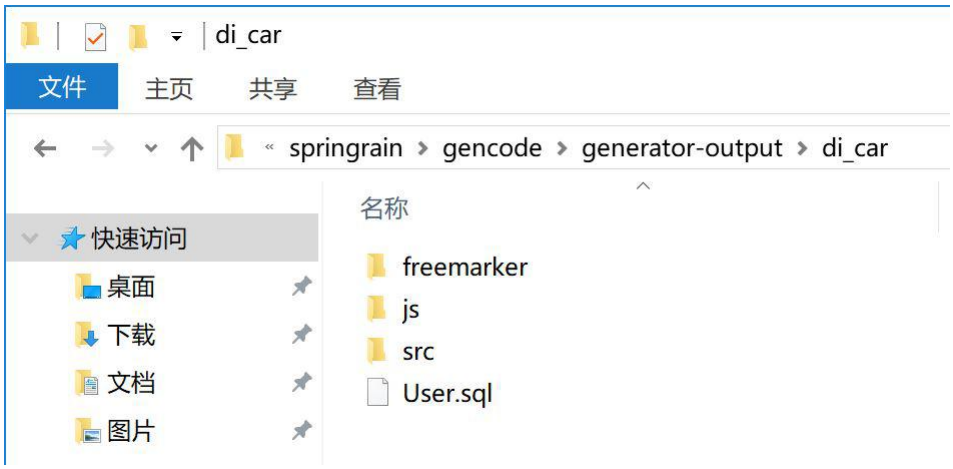
生成器的这

在文件夹中点击运行 gencode/rapid-gen.bat ,输入 gen +表名 回车运行

例如 gen t_user 生成 t_user 表的增删改查



回车生成代码,代码在 gencode/generator-output 目录下,会自动弹出目录



执行代码生成命令(例如: gen t_user)后,生成的代码对应拷贝如下:

di_car/freemarker 对应拷贝到 WebROOT/WEB-INF/freemarker

di_car/js 对应拷贝到 WebROOT/js

di_car/src 对应拷贝到 src

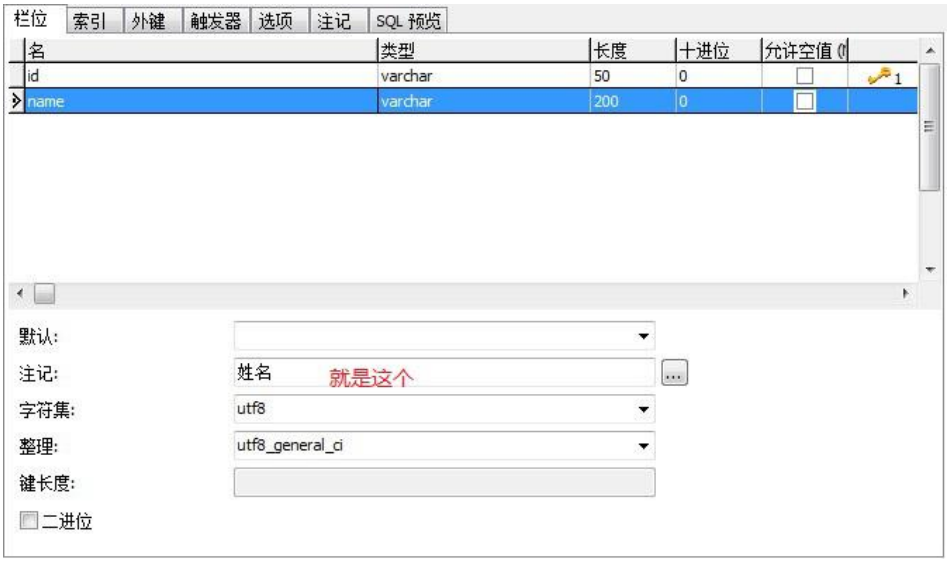
生成的.sql 文件是菜单添加的 sql,一般根据业务修改后在数据库执行.

重点说明:这些代码只是按照默认规则和样式生成的增删改查代码,需要根据实际业务进行修改.

4.3 数据库表列名和页面字段对应生成

代码生成器从数据库取值字段的说明,例如:t_user 表中,字段 name 的注记(备注)是 "姓名",这样就会生成 Entity 和 Freemarker 页面

建议大家维护好数据库中字段的备注说明,这样生成的代码会友好很多.



生成的 Entity 代码

```
/**
 * 姓名
 */
private java.lang.String name;

/**
 * 姓名
 */
public void setName(java.lang.String value) {
    if (StringUtils.isNotBlank(value)) {
        value = value.trim();
    }
    this.name = value;
}

/**
 * 姓名
 */
@WhereSQL(sql = "name=:User_name")
public java.lang.String getName() {
    return this.name;
}
```

生成的 FreeMarker 代码

```
<div class="layui-inline">
    <label class="layui-form-label">名称</label>
    <div class="layui-input-inline">
        <input type="text" name="name" value="${(returnDatas.queryBean.name)!''}" placeholder="请输入名称"
    </div>
</div>
```

5 Freemarker

框架使用了 freemarker 前台渲染,非常简单稳定高效的模版引擎,语法也很简单,看下文档和例子就能上手开发.

Freemarker 生成的列表,修改,查看模版页面,例如生成的 User 页面 userList.html,userCru.html,userLook.html

5.1 查询条件

查询条件的代码为:

```
<div class="layui-inline">
    <label class="layui-form-label">名称</label>
    <div class="layui-input-inline">
        <input type="text" name="name" value="${(returnDatas.queryBean.name)!''}" placeholder="请输入名称"
    </div>
</div>
```

name 和生成的 User 对象属性名称保持一直,后台会自动封装查询条件.*${(returnDatas.queryBean.name)!''}*就是直接从封装对象中取值.后台会自动封装到查询 Bean(默认是 Entity 实体类)中作为查询条件,配合 @WhereSQL 注解自动拼接 SQL 查询条件,下面再详细说明注解.

5.2 字段排序

生成的页面中有以下代码:

```
<!--first_end_no_export-->
    <th id="th_name">姓名</th>
```

类似 <!--first_start_export--> 的 html 注释不要修改和删除,导出会用到.

表头 th 列 id 以 "th_" 开头的列具有排序功能, th_之后的是需要排序的字段,本例中就是需要后台按照"name"字段进行排序,当然也可以是其他,例如添加了别名可以是"th_u.name",这个主要和后台的查询语句有关.

表单的中的隐藏域值会根据 js 操作排序修改,然后提交表单刷新页面.

```
<form class="layui-form layui-form-pane" id="searchForm" action="${ctx}/system/user/list" method="post">
<input type="hidden" name="pageIndex" id="pageIndex" value="${(returnDatas.page.pageIndex)!'1'}" />
<input type="hidden" name="sort" id="page_sort" value="${(returnDatas.page.sort)!'desc'}" />
<input type="hidden" name="order" id="page_order" value="${(returnDatas.page.order)!'id'}" />
```

5.3 按钮权限控制

通过 Shiro 的 shiro.hasPermission 标签,控制按钮是否显示,因为本质上是否有权限访问功能URL,name 的值就是需要控制访问的URL,不具备 URL 访问权限,就不会显示标签内的内容.

```
<li style="float:right;">
    <@shiro.hasPermission name="/system/user/update" >
        <button type="button"   class="layui-btn layui-btn-small" data-action="${ctx}/system/user/update/pre"><i class="layui-icon layu
    </@shiro.hasPermission>
    <@shiro.hasPermission name="/system/user/list/export" >
        <button type="button"   class="layui-btn layui-btn-small"><i class="layui-icon layui-icon-specil">&#xe609;</i>导出</button>
    </@shiro.hasPermission>
    <button type="button"  class="layui-btn layui-btn-warm layui-btn-small"><i class="layui-icon layui-icon-specil">&#xe601;</i>导入</b
    <@shiro.hasPermission name="/system/user/delete" >
        <button type="button"  class="layui-btn layui-btn-danger layui-btn-small"><i class="layui-icon">&#xe640;</i>批量删除</button>
    </@shiro.hasPermission>
</li>
```

5.4 页面按钮事件

按钮事件都经过了封装,主要是处理 token 安全码,一般情况下,可以封装现有实现处理业务,不建议重新实现按钮事件.

6 Controller

框架使用 springmvc， springmvc 非常的强大灵活和高性能,基于注解的方式,rest 风格.....

Controller 的增加/修改(update)和删除(delete)都返回 json 格式的数据,查看(list/look)有页面和 json 两种返回类型./list 是页面,/list/json 是 json;/look 是页面,/look/json 是 json;页面直接封装 json 方法的返回结果,页面和 json 统一接收 org.springrain.frame.util.ReturnDatas 对象

```
/**
 * 列表数据,调用listjson方法,保证和app端数据统一
 *
 * @param request
 * @param model
 * @param user
 * @return
 * @throws Exception
 */
@RequestMapping("/list")
public String list(HttpServletRequest request, Model model,User user)
        throws Exception {
    ReturnDatas returnObject = listjson(request, model, user);
    model.addAttribute(GlobalStatic.returnDatas, returnObject);
    return listurl;
}

/**
 * json数据,为APP提供数据
 *
 * @param request
 * @param model
 * @param user
 * @return
 * @throws Exception
 */
@RequestMapping("/list/json")
@ResponseBody
public  ReturnDatas listjson(HttpServletRequest request, Model model,User user) throws Exception{
    ReturnDatas returnObject = ReturnDatas.getSuccessReturnDatas();
    // ==构造分页请求
    Page page = newPage(request);
    // ==执行分页查询
    List<User> datas=userService.findListDataByFinder(null,page,User.class,user);
        returnObject.setQueryBean(user);
    returnObject.setPage(page);
```

7 Service

7.1 核心实现

每个数据库都会基本的 Service, 例如下图中的 baseSpringrainService, 基本的 Service 需要继承 BaseServiceImpl 这个 Service 基类, 实现 IBaseService 这个基本接口.

```
package org.springrain.system.service;

import java.io.File;

@Service("baseSpringrainService")
public class BaseSpringrainServiceImpl extends BaseServiceImpl implements IBaseSpringrainService {

    @Resource
    IBaseJdbcDao baseSpringrainDao;

    public BaseSpringrainServiceImpl() {
    }
```

```
1  package org.springrain.system.service;
2
3  import org.springrain.frame.service.IBaseService;
4
5
6  public interface IBaseSpringrainService extends IBaseService {
7
8  }
```

继承基本 Service(baseSpringrainService)派生业务 service, 例如 userService

```
package org.springrain.system.service.impl;

import java.io.File;

 * TODO 在此加入类描述
@Service("userService")
public class UserServiceImpl extends BaseSpringrainServiceImpl implements IUserService {

```

IBaseService 已经提供了基本的操作方法, 包含增删改查, 可以看下接口和实现.

7.2 使用

Service 在注入使用时,注入的属性名必须和 Service 注解名字相同.

例如 UserController 使用了 userService,需要保持一致

```
@Service("userService")
public class UserServiceImpl extends BaseSpringrainServiceImpl implements IUserService {
```

```
@Controller
@RequestMapping(value = "/system/user")
public class UserController extends BaseController {
    @Resource
    private IUserService userService;
```

7.3 事务规则

Servcie 方法命名必须严格按照事务拦截的规则,如果一个方法内有事务操作,主方法必须符合事务命名规则.事务拦截配置在:applicationContext-tx.xml

```
    <!-- 基本事务定义,使用transactionManager作事务管理,默认find*方法 没有事务,和动态数据源配合 -->
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <!-- 指定哪些方法需要加入事务，可以使用通配符来只加入需要的方法 -->
            <tx:method name="save*" propagation="REQUIRED"   isolation="READ_UNCOMMITTED"
                rollback-for="Exception" />
            <tx:method name="update*" propagation="REQUIRED"  isolation="READ_UNCOMMITTED"
                rollback-for="Exception" />
            <tx:method name="delete*" propagation="REQUIRED"   isolation="READ_UNCOMMITTED'
                rollback-for="Exception" />
        </tx:attributes>
    </tx:advice>
```

方法名 save,update,delete 开头的会被事务拦截,如果更新操作内没有事务,操作会抛出异常.只要业务操作内有事务操作,执行入口的方法名就必须符合拦截的规则.

例如:

```
public void saveLog(){
        finda();
```

```
        findb();

        save();

}
```

因为有一个 save 操作,所以方法名是 saveLog,依 save 开头.如果不符合规范,会执行抛错

```
/**
 * 检查方法是否有事务
 * @throws Exception
 */
private void checkMethodName()throws NoTransactionException {
    if(isCheckMethodName()){//方法是否具有事务
        try{
            TransactionInterceptor.currentTransactionStatus();
        }catch(NoTransactionException e){
            throw new NoTransactionException("save和update方法,请按照事务拦截方法名书写规范!具体参见:applicationContext-tx.xml");
        }

    }
}
```

7.4 常用方法

7.4.1 查询列表对象

这里只列出部分常用 API 方法,具体还请查看接口方法注释 doc:

```
171  /**
172   * 只有Finder查询语句,返回结果是List<Map>
173   *
174   * @param finder
175   * @return
176   * @throws Exception
177   */
178  public List<Map<String, Object>> queryForList(Finder finder) throws Excep
179
180  /**
181   * 指定返回对象是T,查询结果是 List<T>
182   *
183   * @param finder
184   * @param clazz
185   * @return
186   * @throws Exception
187   */
188  public <T> List<T> queryForList(Finder finder, Class<T> clazz) throws Exc
189
190  /**
191   * 指定分页对象 进行列表查询,返回 List<Map>
192   *
193   * @param finder
194   * @param page
195   * @return
196   * @throws Exception
197   */
198  public List<Map<String, Object>> queryForList(Finder finder, Page page) t
199
200  /**
```

7.4.2查询单个对象

```
/**
 * 只有Finder查询语句,查询一个对象,返回Map对象
 *
 * @param finder
 * @return
 * @throws Exception
 */
public Map<String, Object> queryForObject(Finder finder) throws Exception;

/**
 * 指定返回对象 T,查询一个对象T,返回 T 对象
 *
 * @param finder
 * @param clazz
 * @return
 * @throws Exception
 */
public <T> T queryForObject(Finder finder, Class<T> clazz) throws Exception;

/**
 * T 作为查询的query bean,查询一个对象T,返回T
 *
 * @param entity
 * @return
 * @throws Exception
 */
public <T> T queryForObject(T entity) throws Exception;
```

7.4.3保存

```

    /**
     * 保存一个对象
     *
     * @param <T>
     * @param clazz
     * @return
     */
    public Object save(Object entity) throws Exception;

    /**
     * 批量保存对象
     *
     * @param list
     * @return List
     * @throws Exception
     */
    public List<Integer> save(List list) throws Exception;
```

7.4.4更新

```
/**
 * 更新数据
 *
 * @param sql
 * @param paramMap
 * @return
 * @throws Exception
 */
public Integer update(Finder finder) throws Exception;
/**
 * 更新一个对象,id必须有值
 *
 * @param <T>
 * @param clazz
 * @return
 */
public Integer update(IBaseEntity entity) throws Exception;

/**
 * 更新一个对象,id必须有值,updatenotnull=true 不更新为null的字段,false更新所有字段
 *
 * @param entity
 * @param updatenotnull
 * @return
 * @throws Exception
 */
public Integer update(IBaseEntity entity, boolean onlyupdatenotnull) t

/**
 * 批量更新对象
 *
 * @param list
 * @return
 * @throws Exception
 */
public List<Integer> update(List list) throws Exception;

/**
 * 批量更新对象,id必须有值,updatenotnull=true 不更新为null的字段,false更新所有字段
 *
 * @param list
 * @param updatenotnull
 * @return
 * @throws Exception
 */
public List<Integer> update(List list, boolean onlyupdatenotnull) thro
```

7.4.5删除

```
    /**
     * 根据Id 删除
     *
     * @param id
     * @throws Exception
     */
    public void deleteById(Object id, Class clazz) throws Exception;

    /**
     * 根据ID批量删除
     *
     * @param ids
     * @param clazz
     * @throws Exception
     */

    public void deleteByIds(List ids, Class clazz) throws Exception;

    /**
     * 根据Entity 删除
     *
     * @param IBaseEntity
     *            entity
     * @throws Exception
     */
    public void deleteByEntity(IBaseEntity entity) throws Exception;
```

7.4.6SQL 语句工具类方法

```

    /**
     * 根据查询的queryBean,拼接Finder的 Where条件,只拼接非NULL的值,拼接前Finder中必须包含WHERE,只拼接 and 条件,用于普通查询<br/>
     * 例如:User对象的name属性值为 张三 ,根据name属性的@WhereSQL注解,拼接出来的语句类似:<br/>
     * finder.append(" and name=:User_name").setParam("User_name","张三");
     *
     * @param finder
     * @param o
     * @return
     * @throws Exception
     */
    public Finder getFinderWhereByQueryBean(Finder finder, Object o) throws Exception;

    /**
     * finder清除自身的排序,根据page对象中sort和order 添加order by 排序,一般用于前台传递的自定义排序<br/>
     * 代码类似:<br/>
     * finder.append(" order by ").append(page.getOrder()).append(" ").append(page.getSort());
     *
     * @param finder
     * @param page
     * @return
     */
    Finder getFinderOrderBy(Finder finder, Page page) throws Exception;

```

8 Dao

`IBaseService`的实现只是封装了 Dao 层方法,`IBaseJdbcDao`的数据库操作接口和`IBaseService`接口保持一致.

8.1 一个数据库只有一个 Dao

一个数据库只有一个 Dao,业务通过 Service 实现,例如 userService

每个数据库的 Dao 需要继承抽象类`BaseJdbcDaoImpl`,实现`IBaseJdbcDao`接口.

不同的数据库需要注入不同的数据库方言(IDialect 接口)实现,默认使用 MySQL,所以注入了 mysqlDialect

例如 baseSpringrainDao:

```
20   */
21  @Repository("baseSpringrainDao")
22  public class BaseSpringrainDaoImpl extends BaseJdbcDaoImpl implements IBaseJdbcDao {
23
24      public BaseSpringrainDaoImpl() {
25      }
26
27      /**
28       * demo 数据库的jdbc,对应 spring配置的 jdbc bean
29       */
30      @Resource
31      NamedParameterJdbcTemplate jdbc;
32      /**
33       * demo 数据库的jdbcCall,对应 spring配置的 jdbcCall bean
34       */
35      @Resource
36      public SimpleJdbcCall jdbcCall;
37      /**
38       * mysqlDialect 是mysql的方言,springBean的name,可以参考 IDialect的实现
39       */
40      @Resource
41      public IDialect mysqlDialect;
42
43      @Override
44      public IDialect getDialect() {
45          return mysqlDialect;
46      }
```

8.2 数据库 dialect

只有在查询分页时需要考虑数据库差异,

实现 org.springrain.frame.dao.dialect.IDialect 接口即可.

数据库名称	实现分页函数	说明
SQLServer	ROW_NUMBER()	支持 sql2005+的版本, 不支持 sql2000
Oracle	rownum	
DB2	ROWNUMBER()	
Informix	SKIP FIRST	
PostgreSql	limit	
SQLite3	limit	
Sybase	未实现	

在 dao 中注入实现的 spring bean 即可.

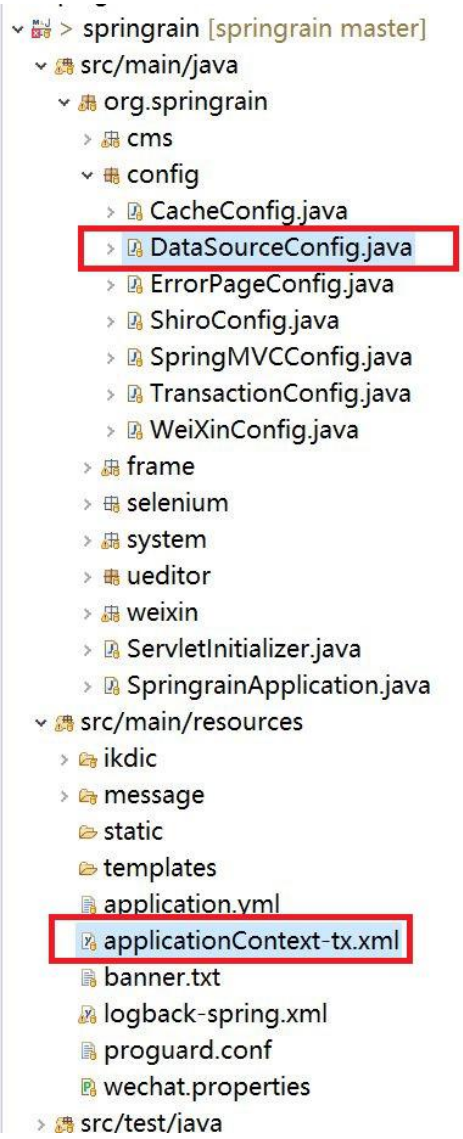
```
/**
 * mysqlDialect 是mysql的方言,springBean的name,可以参考 IDialect的实现
 */
@Resource
public IDialect mysqlDialect;
@Override
public IDialect getDialect() {
    return mysqlDialect;
}
```

8.3 项目使用多个数据库

总共分三步:

1.在 application.yml 中添加数据库的连接字符串和账号密码

2.拷贝改(或者写到同一个配置文件) org.springrain.config.DataSourceConfig 和 applicationContext-tx.xml 的配置文件



例 如 新 增 org.springrain.config.DataSourceConfig2 和 applicationContext-tx2.xml

3.创建基本的 Dao 和 Service

在 dao 中注入配置文件声明的 NamedParameterJdbcTemplate 和 SimpleJdbcCall, 名称和 spirng Bean 一致.

因为默认没有使用 JTA,每个数据库的事务是独立的.所以每个数据库应该有一个独立的根包路径,主要是为了方便 spring 事务扫描,当然如果配置了 JTA,就无所谓了.

9 Entity

Entity 默认包是 org.springrain.frame.entity.BaseEntity 为基础父类,所有的实体 Entity 必须继承 BaseEntity

9.1 使用的注解

@Table 为映射的表名

@TableSuffix 分表后缀.值为获取分表后缀的字段,在 save 或者 update 对象操作时,可以根据对象的属性值确定分表的后缀.参见 org.springrain.demo.entity.AuditLog

@NotLog 不记录日志

@Id 为主键 ID,放在字段的 get 方法上,可以支持 UUID 和自增,默认为 UUID

@Transient 放在字段的 get 方法上,标示数据库不存在的属性

@WhereSQL,拼装 sql 的 where 条件,对于简单查询,enity 可以直接作为 querybean 作为查询条件.

最后通过 org.springrain.frame.dao.BaseJdbcDaoImpl.getFinderWhereByQueryBean(Finder, Object)

拼装 where 条件,Object 形参就是 QueryBean ,默认为 Entity.

通过 org.springrain.frame.dao.BaseJdbcDaoImpl.getFinderOrderBy(Finder, Page)可以拼接前台界面拼接的 order by

```
@WhereSQL(sql="operatorName<:Auditlog_operatorName")
@WhereSQL(sql="operationType like :%Auditlog_operationType%")
```

Entity 的属性名需要和数据库完全一致,也可以在拼写 sql 语句时起别名.

@PKSequence,处理数据库 sequence 的主键自增,这个注解必须和@Id 配合使用,当 Number 类型的主键值为 null 时,会取值@PKSequence 下面的是 oracle 的序列

```
@PKSequence(name="test.nextvalue")
@Id
@WhereSQL(sql="id=:testTable_id")
public java.lang.Integer getId() {
```

最终执行的语句 类似如下

```
insert into testtable(id ,name) values(test.nextvalue,"testName");
```

9.2 设置别名

当使用 Finder 拼装语句时,可以使用 `setFrameTableAlias` 设置 Entity 作为 QueryBean 的 SQL 别名.

10 Finder

Finder 是封装 SQL 查询的实体对象,框架中所有的业务 SQL 语句最终都是通过 Finder 执行的.Finder 的方法返回值基本都是 Finder 本身,方便链式调用.

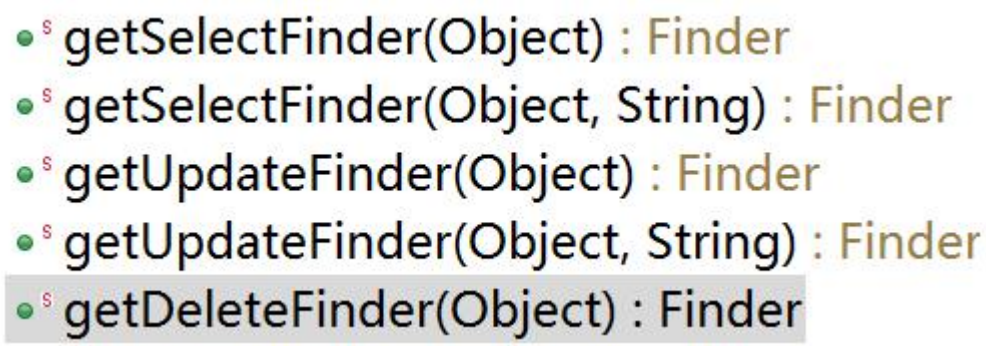
10.1 直接写 SQL 语句

最灵活的方式,可以实现你的任何 sql 语句编写,不允许显示写表名,必须使用 Finder.getTableName(Object)方法获取表名!!!!

```
Finder finder=new Finder("SELECT * FROM ").append(Finder.getTableName(User.class)).append(" WHERE id=:id ");
finder.setParam("id", "admin");
```

10.2 获取具体操作的 Finder 对象

快捷方法,能够根据实体类获取 Finder 对象



例如：

Finder.getSelectFinder(User.class)

Finder.getSelectFinder(User.class, “id, name”)

10.3 like 的写法

数据库的 like 查询在框架有两种实现:

◆ 通过 SQL 语句实现 like

```
String operationType="abc";
finder.append(" and operationType like :operationType").setParam("operationType", "%"+operationType+"%");
```

◆ 通过@WhereSQL 注解实现 like 功能

```
@WhereSQL(sql="operationType like :%Auditlog_operationType%")
```

这种方法底层也是转成了 finder 的 SQL 语句进行执行的,只是使用 QueryBean 时会方便点.

10.4 使用范例

test.SpringTest.testSelectUser()

```
@Test
public void testSelectUser() throws Exception{

    //框架默认Entity做为QueryBean,也可以自己定义QueryBean
    User queryBean=new User();
    queryBean.setId("admin");

    //初始化Finder,并为User取别名 u
    //Finder finder=Finder.getSelectFinder(User.class," u.*").append("WHERE  1=1 ");
    Finder finder=new Finder("SELECT u.* FROM ").append(Finder.getTableName(User.class)).append(" u WHERE 1=1 ");
    //finder.append(" and u.id=:userId ").setParam("userId", "admin");

    //设置别名为u
    queryBean.setFrameTableAlias("u");

    //拼接queryBean作为查询条件
    userService.getFinderWhereByQueryBean(finder, queryBean);
    //查询第一页
    Page page=new Page(1);
    //分页查询List<User>集合对象
    List<User> list = userService.queryForList(finder,User.class,page);

    System.out.println(list);
}
```

11 Lucene

在 Entity 上使用@LuceneSearch 注解标识这个实体类会使用 Lucene 进行全文检索. 底层在对这个类对象进行更新操作时会自动更新 Lucene 索引文件, 注意只能监控到针对对象的更新操作, 通过 Finder 的语句更新无法监控, 所以不建议使用@LuceneSearch 自动处理索引.

实体类上不使用@LuceneSearch 注解, 可以通过 LuceneUtils 手动控制索引的创建和更新. 建议这种方式, 比较灵活.

字段的 get 方法上使用@LuceneField 标识字段进行索引

```java
/**
 * 标记可以用于Lucene搜索,根据类名创建索引
 *
 * @copyright {@link weicms.net}
 * @author springrain<9iuorg@gmail.com>
 * @version 2013-03-19 11:08:15
 * @see org.springrain.frame.annotation.LuceneField
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface LuceneField {
    // 是否分词,仅支持字符串,主键强制不分词
    boolean stringTokenized() default true;

    // 是否进行lucene排序字段,仅支持数值和日期类型
    boolean numericSort() default true;

    //FacetField,暂未实现 facet,这样的场景建议换solr了
    //boolean luceneFacet() default false;

    // 字段是否保存,请谨慎修改
    boolean luceneStored() default true;

    // 字段是否索引,只有索引才能作为查询条件,请谨慎修改
    boolean luceneIndex() default true;
}
```

通过 `LuceneFinder` 控制查询条件. 具体参见 test.LuceneTest

12 安全

12.1 SQL 注入

后台主要的安全问题是 SQL 注入, 处理方式是禁止 sql 语句中包含单引号(‘), 避免 SQL 中拼接字符串参数, 规避 SQL 注入风险, 具体实现 org.springrain.frame.util.Finder.getSql()方法, 因为所有的 SQL 都是通过 Finder 执行, 可以全局判断 SQL 语句, 发现即中断执行.

Finder 中预留了特殊情况的处理开关, 通过查看方法引用, 就可以查看到特殊情况, 若无必要, 会责令修改.

12.2 XSS 防护

XSS 是最常见的 js 攻击方式,具体实现百度.....

针对 XSS 防护,最基本的是特殊字符转义和过滤.一般有,转义后保存和渲染时转义,我们使用了渲染时转义.

Freemarker(2.3.25+版本)使用<#ftl output_format="HTML" auto_esc=true>进行特殊字符转义.

AJAX 请求,就绕过了 freemarker,扩展 jackson 接口,转义特殊符号,

参见:org.springrain.frame.util.FrameObjectMapper,转义特殊字符

```
public FrameObjectMapper(){

        //为null 的不转换
        // this.setSerializationInclusion(JsonSerialize.Inclusion.NON_NULL);
    this.setSerializationInclusion(JsonInclude.Include.NON_NULL);
        //支持 属性不对应
    this.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES , false);
        //为bean 为null时不报异常
    this.configure(SerializationFeature.FAIL_ON_EMPTY_BEANS,false);
        //键 支持 不带 双引号 ""
    this.configure(JsonParser.Feature.ALLOW_UNQUOTED_FIELD_NAMES, true);
        //过滤敏感字符
    this.getFactory().setCharacterEscapes(new HTMLCharacterEscapes());
```

富文本编辑是注入高发区,需要过滤用户输入的 html 代码标签,我们使用 jsoup,具体实现:

org.springrain.frame.util.InputSafeUtils.filterTextContent(String)

```
  private final static Whitelist user_content_filter = Whitelist.relaxed();

  static {
      user_content_filter.addTags("embed","object","param","span","div");
      user_content_filter.addAttributes(":all", "style", "class", "id", "name");
      user_content_filter.addAttributes("object", "width", "height","classid","codebase");
      user_content_filter.addAttributes("param", "name", "value");
      user_content_filter.addAttributes("embed", "src","quality","width","height","allowFullScreen",'
  }

  /**
   * 对用户输入内容进行过滤,用于普通的文本字段
   * @param html
   * @return
   */
  public static String filterTextContent(String text) {
      if(StringUtils.isBlank(text)){
          return text;
      }
      text=StringEscapeUtils.escapeHtml4(text);
      return text;
  }
```

12.3 CSRF 防护

依然自行百度.....

使用 token 机制规避 CSRF 请求,账号登录成功过时,在 session 中保存 token,添加到用户的请求链接上,使用拦截器进行验证,前台封装了 js 方法,只需要调用框架提供的 js 方法即可,不需要关注实现细节.

参见:org.springrain.frame.shiro.FrontUserFilter

```
//设置tokenkey
String springraintoken="f_"+SecUtils.getUUID();
session.setAttribute(GlobalStatic.tokenKey, springraintoken);
model.addAttribute(GlobalStatic.tokenKey,springraintoken);
```

```
Object obj=req.getSession().getAttribute(GlobalStatic.tokenKey);
String token=obj.toString();

String userToken=req.getParameter(GlobalStatic.tokenKey);
if(token.equals(userToken)){
    return true;
}

request.setAttribute(GlobalStatic.errorTokentoURLKey, GlobalStatic.errorTokentoURL);
```

13 性能

13.1 缓存

缓存是系统性能的生死线.....

一般缓存系统的结构是 Map<String,Map<String,Object>>

shiro 的缓存配置 org.springrain.config.ShiroConfig,有单机缓存和集群两种方式

```
172
173  /**
174   * 单机session
175   * @return
176   */
177  @Bean("shiroCacheManager")
178  public CacheManager shiroCacheManager() {
179     MemoryConstrainedCacheManager cacheManager=new MemoryConstrainedCacheManager();
180     return cacheManager;
181  }
182
183  /**
184   * session 集群
185   * @return
186   */
187  /*
188  @Bean("shiroCacheManager")
189  public CacheManager shiroCacheManager() {
190     ShiroRedisCacheManager cacheManager=new ShiroRedisCacheManager();
191     cacheManager.setRedissonClient(redissonClient);
192     return cacheManager;
193  }
194  */
195
```

Spring 的缓存配置 org.springrain.config.CacheConfig,有单机缓存和集群两种方式,shiro 缓存只在 shiro 相关的 API 实现,业务缓存使用 spring 缓存.单机和集群配置方式和 shiro 保持同步

```
/**
 * 基于内存的cacheManager
 *
 * @return
 * @throws IOException
 */

@Bean("cacheManager")
public CacheManager cacheManager() {
    CacheManager cacheManager = new ConcurrentMapCacheManager();

    return cacheManager;
}

// --------基于redis的cacheManager 开始--------//

/**
 * 基于redis的cacheManager,使用redisson客户端
 *
 * @return
 * @throws IOException
 */
/*
@Bean("cacheManager")
public CacheManager cacheManager() {
    return new RedissonSpringCacheManager(redissonClient());
}

@Bean("redissonClient")
public RedissonClient redissonClient()  {

    // 连接超时时间
    int connectTimeOut = 10000;
    // 重试次数
    int retryAttempts = 6;

    if (StringUtils.isBlank(redisHostPort)) {
        return null;
    }
```

可以使用 spring 的@Cacheable 和@CacheEvict 注解处理缓存,

目前建议手动控制,不使用 Spring 的注解控制缓存,手动控制更灵活.

已经在底层 org.springrain.frame.service.BaseServiceImpl 实现,每个继承的 service 都可以直接使用

```
44
45      /**
46       * 获取Cache
47       * @param cacheName
48       * @return
49       * @throws Exception
50       */
51      public Cache getCache(String cacheName) throws Exception;
52
53      /**
54       * 查找cache的对象
55       * @param cacheName
56       * @param key
57       * @param clazz
58       * @return
59       * @throws Exception
60       */
61      public <T> T getByCache(String cacheName,String key,Class<T> clazz)throws Exception;
62      /**
63       * 设置缓存对象
64       * @param cacheName
65       * @param key
66       * @param obj
67       * @throws Exception
68       */
69      public void putByCache(String cacheName,String key,Object value)throws Exception;
70
71
72      /**
73       * 清理缓存
74       * @param cacheName
75       * @throws Exception
76       */
77      public void cleanCache(String cacheName)throws Exception;
78      /**
79       * 清理缓存下的一个key
80       * @param cacheName
81       * @throws Exception
82       */
83      public void evictByKey(String cacheName,String key)throws Exception;
84
```

针对分页的 Page 对象,定义了专门的 API 方法,用于同时缓存 Page 对象

```
87
88      /**
89       * 查找cache的列表查询结果对象,主要用于列表的缓存.查询出缓存结果并对page对象赋值,参数Page 可以是 Page page=null,这样传入
90       * @param cacheName
91       * @param key
92       * @param clazz
93       * @return
94       * @throws Exception
95       */
96      public <T> T getByCache(String cacheName,String key,Class<T> clazz,Page page)throws Exception;
97      /**
98       * 设置缓存列表查询的缓存对象,并对Page进行缓存.
99       * @param cacheName
100      * @param key
101      * @param obj
102      * @throws Exception
103      */
104     public void putByCache(String cacheName,String key,Object value,Page page)throws Exception;
105
106     /**
107      * 清理缓存下的一个key,并清除缓存的page,page对象可以为null
108      * @param cacheName
109      * @throws Exception
110      */
111     public void evictByKey(String cacheName,String key,Page page)throws Exception;
112
```

13.2 数据库读写分离

13.2.1 事务粘性

一个事务内的操作在同一个数据库,避免读写分离的延迟问题.推荐!!!

如果有事务就使用写库操作,没有事务就使用读库操作.

修改 DataSource 的实现配置,使用 `TransactionDataSourceRouter` 做为 DateSource,注入 `dataSourceWrite` 和 `dataSourceRead` 两个标准的 DateSource,核心是基于 Spring 的 `AbstractRoutingDataSource` 实现数据源的动态切换.

```
96
97
98      <!-- 根据事务隔离级别做读写分离,有事务就写,无事务就是读 -->
99      <!--
100     <bean id="dataSource" class="org.springrain.frame.dao.TransactionDataSourceRouter">
101        <property name="targetDataSources">
102          <map>
103            <entry key="dataSourceWrite" value-ref="dataSourceWrite"/>
104            <entry key="dataSourceRead"  value-ref="dataSourceRead"/>
105          </map>
106        </property>
107        <property name="defaultTargetDataSource" ref="dataSourceRead"/>
108     </bean>
109     -->
110
111
```

使用 `FrameDataSourceTransactionManager` 替换默认的事务管理器.

```
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"></property>
</bean>

<!-- 处理读写分离,需要复写 DataSourceTransactionManager -->

<!--
<bean id="transactionManager" class="org.springrain.frame.dao.FrameDataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"></property>
</bean>
-->
```

13.2.2 读写强制切换

不考虑事务,强制读写切换,理论性能较好,风险较高,不推荐.

是基于底层 Dao,注入读写的 NamedParameterJdbcTemplate 实现.

```
    */
    public abstract NamedParameterJdbcTemplate getJdbc();

    /**
     * 抽象方法.每个数据库的代理Dao都必须实现.在多库情况下,用于区分底层数据库的连接对象,调用数据库的函数和存储过程.</br>
     * 例如:demo 数据库的代理Dao org.springrain.demo1.dao.BasedemoDaoImpl 实现返回的是spring的bean
     * jdbcCall.</br>
     * datalog 数据库的代理Dao org.springrain.demo2.dao.Basedemo2DaoImpl
     * 实现返回的是spring的beanjdbcCall_demo2.</br>
     *
     * @return
     */
    public abstract SimpleJdbcCall getJdbcCall();

    /**
     * 读写分离的读数据库
     *
     * @return
     */
    public NamedParameterJdbcTemplate getReadJdbc() {
        return getJdbc();
    }

    /**
     * 读写分离的写数据库
     *
     * @return
     */
    public NamedParameterJdbcTemplate getWriteJdbc() {
        return getJdbc();
    }
```

14 业务实现流程

基于 CMS 核心,使用标签实现的三层管理体系

/system 开头的是平台总管理后台

/s 开头的是站长管理后台

/f 开头的是前台用户

目前计划 /uc 开头的是前台会员登陆后的用户中心

三层平台有三个对应的拦截器.

权限依次是 system > s > f

shiro 截图如下.

```
</property>
<!-- 声明自定义的过滤器 -->
<property name="filters">
    <map>
        <!-- 访问日志记录的过滤器 -->
        <entry key="framefwlog" value-ref="framefwlog"></entry>
        <!--权限校验的过滤器 -->
        <entry key="frameperms" value-ref="frameperms"></entry>

         <!-- 前台用户过滤器 -->
        <entry key="frontuser" value-ref="frontuser"></entry>
         <!-- 网站后台用户过滤器 -->
        <entry key="siteuser" value-ref="siteuser"></entry>
         <!-- 后台用户过滤器 -->
        <entry key="systemuser" value-ref="systemuser"></entry>

        <!-- 踢出上个账户的过滤器 -->
        <entry key="keepone" value-ref="keepone"/>
         <!-- 静态化 过滤器 -->
        <entry key="statichtml" value-ref="statichtml"/>

         <!-- 防火墙 过滤器 -->
        <entry key="firewall" value-ref="firewall"/>


        <!-- 微信登录验证过滤器 -->
        <entry key="wxmpautologin" value-ref="wxmpautologin"/>


    </map>
</property>
```

所有的页面跳转连接都配置在 cms_link 表,通过 jump 方法进行查找跳转地址,例如:

```
@Controller
@RequestMapping("/f/pc/{siteId}")
public class FrontPcController extends FrontBaseController {
    /**
     * 映射首页页面
     * @throws Exception
     * */
    @RequestMapping("/index")
    public String index(@PathVariable String siteId,HttpServletRequest request,Model model)
        return jump(siteId, siteId, OrgType.pc.getType(), request, model,Enumerations.CmsLi
    }

    /**
     * 映射栏目页面
     * @throws Exception
     * */
    @RequestMapping("/{businessId}")
    public String channel(@PathVariable String siteId,@PathVariable String businessId,HttpS
        ReturnDatas returnDatas = ReturnDatas.getSuccessReturnDatas();
        Page page = newPage(request);
        returnDatas.setPage(page);
        model.addAttribute(GlobalStatic.returnDatas, returnDatas);
        Integer modelType=Enumerations.CmsLinkModeType.前台栏目.getType();
        if(businessId.startsWith("c_")){
            modelType=Enumerations.CmsLinkModeType.前台内容.getType();
        }
        return jump(siteId, businessId, OrgType.pc.getType(), request, model,modelType);
    }
}
```

前台页面尽量通过标签实现,标签输出不再使用 tag_bean,默认使用 data_+标签名称,例如:data_cms_link

15 规范约定

15.1 不允许改动 frame 包下文件

15.2 SQL 语句中不允许拼接前台参数,必须使用占位符

错误写法!SQL 注入啊,大哥!!!!前台数据都是骗人的啊!!!

```
String name =request.getParameter("name");
Finder finder=new Finder("SELECT * FROM ").append(Finder.getTableName(User.class)).append(" WHERE name="+name+"");
```

正确写法:

```
String name =request.getParameter("name");
Finder finder=new Finder("SELECT * FROM ").append(Finder.getTableName(User.class)).append(" WHERE name=:name ");
finder.setParam("name", name);
```

15.3 不允许直接拼接表名，只能使用 Finder 工具类获取表名

手动拼写容易写错,而且还有数据库大小写的问题,如果数据库表名做了修改,后果是灾难性的.

Finder 工具类更方便书写,IDE 也会有提示校验,安全稳定效率好.

15.4 Controller 编写规范

原则上不允许新增新的逻辑方法，以代码生成为准。

只能返回 String，void，ReturnDatas。

所有的数据库语句写在 Service，Controller 不允许写 SQL 语句

/json,/pre,/more 会被截取然后在判断权限,/update/pre 的权限等同/update

/**/ajax/**=user,URL 中包含/ajax/的路径只要是登陆用户都可以访问,特殊的 URL 配置权限过滤即可.

导入 Excle 模板

name	age
姓名	年龄
张三	18
李四	19

,第一行是实体类的属性名,第二行是用户能看懂的中文名,第三行是实际数据.调用 Service 的方法接口

```
/**
 * 导入Excle文件
 * @param excel
 * @param clazz
 * @return
 * @throws Exception
 */
public <T> String saveImportExcelFile(File excel,Class<T> clazz)throws Exception;


/**
 * 导入Excle文件
 * @param excel
 * @param clazz
 * @return
 * @throws Exception
 */
public <T> String saveImportExcelFile(File excel,Class<T> clazz,boolean istest)throws Exception;



/**
 * Excel 导入时会循环调用该方法
 * @param entity
 * @param index TODO
 * @param listTitle TODO
 * @param issave TODO
 * @return
 * @throws Exception
 */
public String saveFromExcel(Object entity, int index, boolean istest, List<String> listTitle) throws Exception;
```

导出是直接使用了查询的列表模板,通过 HTML 注释区别特殊情况.实现了查询和导出的数据格式统一.

15.5 Service 编写规范及常用方法

业务查询方法 增:save 开头,删:delete 开头,改:update 开头,查:find 开头.

尽量避免使用数据的特殊函数,例如日期操作的函数.

queryForList

queryForObject

update(Finder finder)

update(List list)

save(Object object)

save(List list)

```
/**
 * 更新一个对象
 * @param <T>
 * @param clazz
 * @return
 */
public  Integer update( IBaseEntity entity) throws Exception;


/**
 * 更新一个对象,id必须有值,updatenotnull=true 不更新为null的字段,false更新所有字段
 * @param entity
 * @param updatenotnull
 * @return
 * @throws Exception
 */
   public Integer update(IBaseEntity entity,boolean onlyupdatenotnull) throws Exception;
```

15.6 不允许复写 Dao，一个数据库只能有一个 Dao

15.7 不允许手动编写分页函数和其他特定函数

关于分页,如果确定查询一个对象,可以使用 queryForObject.

如果是从 list 中想取出第一条,不要手动编写 limit 0,1 可以使用 Page 对象分页查询,page.setPageIndex(1);page.setPageSize(1);就是查询列表的第一页第一条.这样兼容性好,可以切换到不同的数据库,如果是 limit 写死代码,只能使用 mysql 了!

除了常用的聚合函数:sum,count,max,min,avg 之外,原则上不再允许使用其他数据库函数.特别是日期相关的函数,绝大部分情况都可以使用 java 代码配合实现.因为数据库资源是最宝贵的,而函数都是很耗数据库资源的,能 java 代码实现的就不要麻烦数据库了!