# Spring介绍

Spring是一个开源框架，是为了解决企业应用开发的复杂性而创建的;

功能：使用基本的SpringBean代替EJB，并提供了更多的企业应用功能,适用于任何Java应用,起到承上启下;

简言之-->Spring是一个轻量级的控制反转(IOC)和面向切面(AOP)的容器框架。

核心：

IOC---控制反转——Spring通过一种称作控制反转(IOC)的技术促进了松耦合。当应用了IoC，某一接口的具体实现类的选择控制权从调用类中移除， 转交给第三方裁决。

AOP---Spring提供了面向切面编程的丰富支持，允许通过分离应用的业务逻辑与系统级服务(例如审计(auditing)和事务()管理)

进行内聚性的开发。应用对象只实现它们应该做的——完成业务逻辑——仅此而已。它们并不负责其它的系统级关注点，例如日志或事务支持。

注意本文是基于spring3.0.6 版本描述的，某些内容可能不适于spring 4.x。

## Spring的模块

1 Core IOC实现 核心容器

2 AOP 面向切面编程

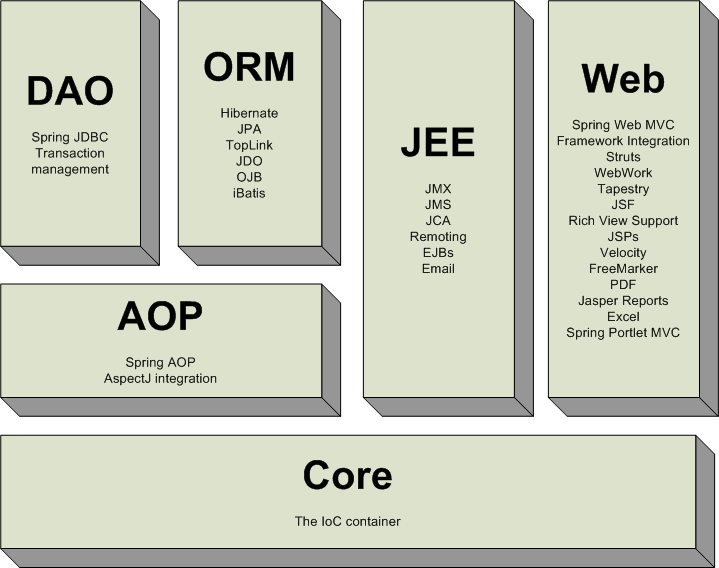
3 DAO 对JDBC的支持

4 ORM 对ORM框架的支持--->hibernate,JPA,TopLink,JDO,OJB,ibatis

5 JEE 对JavaEE 13个技术规范的支持

6 WEB 对WEB框架的支持

7 SpringMVC



## spring 的版本：

现有的spring版本有2.x 3.x 4.x 5.x

3.0之后对应的是JDK\_1.5，才开始支持泛型，所以现在2.x的版本基本已经不再使用了。现在主流的版本是3.x 和4.x。本文以3.0.6为基础进行学习。

Spring3.0的特性

必须使用Jdk1.5以上。

支持根据BeanID获取Bean实例，支持泛型。

支持根据类名－接口名获取实例。

支持表达式语言。#{…}

支持新的线程池

直接支持任务调度

## Spring jar包组成

spring-context.jar：提供IoC高级功能，JNDI、EJB的支持等

spring-aop.jar：Spring的AOP框架

spring-dao.jar：提供了对数据库的访问抽象，提供了对JDBC和数据库事务的支持

spring-orm.jar：提供了对Hibernate, JDO, iBATIS和OJB的支持

spring-web.jar：提供了对web的支持

spring-webmvc.jar：Spring自己的Web框架

spring-mock.jar：提供了通过Mock方式进行测试的包

org.springframework.beans所有应用都要用到的，它包含访问配置文件、创建和管理bean，以及进行Inversion of Control / Dependency Injection（IoC/DI）操作相关的所有类

org.springframework.core 包含Spring框架基本的核心工具类，Spring其它组件要都要使用到这个包里的类，是其它组件的基本核心

org.springframework.asm Spring独立的asm程序

org.springframework.expression Spring表达式语言

org.springframework.context.support Spring提供在基础IoC功能上的扩展服务，此外还提供许多企业级服务的支持，如邮件服务、任务调度、JNDI定位、EJB集成、远程访问、缓存以及各种视图层框架的封装等

commons-logging Spring日志功能

# 入门使用

## 使用Spring基本步骤

步骤一、导入jar包 6 个核心包

步骤二、spring的配置文件 src/applicationContext.xml 文档校验 schema

建议使用MyEclipse生成或在官方提供的样例

步骤三、在applicationContext.xml文档中添加要生成对象的bean

<bean id="User" class="com.project.entity.User"></bean>

步骤四、获得JavaBean

//1 加载spring的配置文件 利用ApplicationContext(spring的上下文对象)

ApplicationContext app = new ClassPathXmlApplicationContext("applicationContext.xml");

//2 根据bean的id获取bean

User user = app.getBean("User",User.class);

## Bean工厂

关于可以实现获得JavaBean的接口--->ApplicationContext 和 BeanFactory

ApplicationContext是 BeanFactory的子接口，比BeanFactory拥有更多的功能，例如加载 properties 等文件。

另外，创建bean对象的时间点不同，ApplicationContext 创建bean对象 ，是在加载完spring的配置文件，

spring就会把配置文件中的所有的bean配置都创建对象 并且把这些对象放入spring的核心容器中

BeanFactory 是在 getBean()时才创建对象指定需要的对象。

## ApplicationContext创建bean的时机

<bean id=*"userEntity"* class=*"com.zx.entity.User"* lazy-init=*"default"* scope=*"prototype"*></bean>

### 创建时机

spring默认是启动完spring的时候，创建bean；

### lazy-init属性

在spring的配置元素中一个属性，lazy-init="",它的取值有三个，default/false/true

default/false，是在启动的时候，创建bean；

true，是在getBean时才要去创建。

### lazy-init创建bean的优缺点

优点：在第一种情况下可以在启动spring容器的时候，检查spring容器配置文件的正确性，如果再结合tomcat,

如果spring容器不能正常启动，整个tomcat就不能正常启动。

缺点：但是这样的缺点是把一些bean过早的放在了内存中，如果有数据，则对内存来是一个消耗

在第二种情况下，可以减少内存的消耗，但是不容易发现错误

### 多例

Spring默认创建对象是单例的，只有一种情况：在context.getBean时才创建对象

spring的bean中的scope

1、由spring产生的bean默认是单例的

2、可以在spring的配置文件中，scope的值进行修改="singleton/prototype/request/session/global session"

singleton是单例模式，prototype是多例模式

* singleton：默认的scope，每个scope为singleton的bean都会被定义为一个单例对象，该对象的生命周期是与Spring IOC容器一致的（但在第一次被注入时才会创建）。
* prototype：bean被定义为在每次注入时都会创建一个新的对象。
* request：bean被定义为在每个HTTP请求中创建一个单例对象，也就是说在单个请求中都会复用这一个单例对象。
* session：bean被定义为在一个session的生命周期内创建一个单例对象。
* application：bean被定义为在ServletContext的生命周期中复用一个单例对象。
* websocket：bean被定义为在websocket的生命周期中复用一个单例对象。

### spring容器对象的生命周期：

1、spring容器创建对象

2、执行init方法

3、调用自己的方法

4、当spring容器关闭的时候执行destroy方法

思考：当scope为"prototype"时，spring容器是否调用destroy方法？是。

### 关于继承

1、如果一个类在spring配置文件中，但是不想让整个类创建对象，则用abstract="true"

2、如果让一个子类拥有父类的属性，则parent="commonDao"

# 注入值

注意：Spring创建对象，默认是单例的。修改这个特性：<bean id="" class="" scope="prototype">

## set方法注入

需要在实体类中有对应的set方法。

<bean id="UserEntity1" class="com.project.entity.User" scope="prototype">

<property name="id" value="100"></property>

<property name="name" value="张三"></property>

<property name="age" value="20"></property>

</bean>

## 构造器方法注入

需要在实体类中有对应的构造器，index是其参数的位置，从0开始

注入的值必须和对应的JavaBean中的构造器的传参的个数一个，限制条件太多，使用较少

<bean id="UserEntity" class="com.project.entity.User" scope="prototype">

<constructor-arg index="0" value="100"></constructor-arg>

<constructor-arg index="1" value="张三"></constructor-arg>

<constructor-arg index="2" value="20"></constructor-arg>

</bean>

## 接口注入（在业务中，经常使用）

解除耦合，例如，在使用service层的使用，需要调用持久层的Javabean，为了方便以后的修改，一般，是将持久层的创建为属性；

步骤一：创建service的时候，将需要的持久层设置为属性，必须有gatter\setter

public class UserServiceImpl implements UserService {

private UserDao ud ;

public UserDao getUd() {

return ud;

}

public void setUd(UserDao ud) {

this.ud = ud;

}

public void add(User user) {

ud.add(user);

}

}

步骤二：配置JavaBean,将需要的持久层属性，它指向UserDao接口的一个实现类，静候需要修改这个业务的持久层的时候们只需要改变指向的

实现类即可,其实这种接口的注入，也是set方法注入的进一步使用

<bean id="UserService" class="com.project.service.UserServiceImpl">

<property name="ud" ref="UserHibernate" ></property>

<property name="ud" ref="UserJDBC" ></property><!-- 业务改变修改实现类-->

</bean>

UserDao接口的实现类

<bean id="UserJDBC" class="com.project.dao.UserJDBCDaoImpl"></bean>

<bean id="UserHibernate" class="com.project.dao.UserHibernateDaoImpl"></bean>

## bean中id和name的区别

最近在开发项目的时候发现spring的BeanFactory创建bean的时候有时候有2种不同的配置，

即<beanid="test"class="com.TestImpl">与<bean name="test"class="com.TestImpl">，下面一起分析一下这2种配置的区别：

1、命名规范。id属性命名必须满足xml命名规范，而name属性命名可以不用遵守。

不论是id属性还是name属性在创建bean的时候不遵守命名规范会抛出异常。

2、唯一性。id属性命名必须严格遵守唯一原则，否则会报错；name属性命名可以不必遵守唯一原则，

在创建实例对象的时候后面一个bean会覆盖前面一个bean。开发的时候劲量都使用id属性配置bean，严格的检测能让程序更加良好。

## 注入集合

### 实体类的属性有集合属性

public class Collections {

private List<String> list;

private Set<String> set;

private Map<String, String> map;

}

### 配置JavaBean

<bean id="collections" class="com.project.entity.Collections">

<property name="list">

<list>

<value>111</value>

<value>222</value>

<value>333</value>

</list>

</property>

<property name="set">

<set>

<value>aaa</value>

<value>bbb</value>

<value>ccc</value>

</set>

</property>

<property name="map">

<map>

<entry key="1" value="a"></entry>

<entry key="2" value="b"></entry>

<entry key="3" value="c"></entry>

</map>

</property>

</bean>

### 输出值

map,使用entry，一个entry内存储的是一个map的键值对 ，其中Map.Entry,是map的内部接口

@Test

public void test(){

ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");

Collections co = (Collections)ac.getBean("collections", Collections.class);

for (String s1 : co.getList()) {

System.out.println(s1);

}

for (String s2 : co.getSet()) {

System.out.println(s2);

}

for (Entry<String, String> en : co.getMap().entrySet()) {

System.out.println(en.getKey()+"\t"+en.getValue());

}

}

# IOC 和 DI

## IOC （Inversion Of Control 控制反转）

通俗话解释：

在使用spring之前，对象都是在程序中new的方式来创建 创建对象的控制权在

程序 哪里使用哪里new.在使用spring之后，创建对象的工作交给了spring，

创建对象的控制权交给spring

书面解释：

控制反转就是应用程序本身不负责依赖对象的创建及维护，依赖对象的创建及维护是由外部容器负责。

这样控制权就由应用程序本身转移到了外部容器，控制权的转移就是所谓的反转。

## DI（Dependency Injection 依赖注入）

就是在运行期，由外部容器动态的将依赖对象注入到组件中 。或是在运行期，由外部容器决定具体执行的类代码

广义上来说：

class A{

B b;

}

class B{

}

在A类中 使用了B类，那么就说 A 依赖B

准确来说：

A 中 使用B 如果B 作为 A的成员变量 关联关系

A 中 使用 B 如果B作为A中的方法的参数 或方法的局部变量 依赖关系

依赖关系靠spring注入而不是自己去new

## 自动装配

自动去查找与属性中对应的，byName是找bean中，id是User的；byType是找bean中，与User属性的类型一样的bean

<bean auto-wire="byName/byType">

<property name="User" ref="UserEntity">

</bean>

或

<beans default-autowire ="byName/byType">

尽量不用自动装配,是一种不稳定的开发技术。

# 资源加载

## 加载properties文件

例如：在test.properties文件中，有属性 test.key=abc

步骤一：将外部属性文件引入spring的配置文件中，引入之后才可以使用

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations">

<list>

<value>classpath:test.properties</value>

</list>

</property>

</bean>

步骤二；使用属性文件中的属性，在spring的配置文件中使用 ${key}

例如：

<bean id="UserEntity" class="com.project.entity.User">

<property name="name">

<value>${test.key}</value>

</property>

</bean>

## 多个spring.xml文件

1 new ClassPathXmlApplicationContext(new String[]{"","","",""}),加载配置文件的时候，同时加载多个配置文件

String[]{"applicationContext.xml","applicationContext-user.xml","applicationContext-project.xml"}

2 使用通配符

new ClassPathXmlApplicationContext(new String[]{"",""}),加载配置文件的时候，同时加载多个配置文件

String[]{"applicationContext.xml","applicationContext-\*.xml"}\*代表其他的配置文件

3 在applicationContext.xml 中 使用 <import> 导入其他文件

<import resource="applicationContext-user.xml"/>

# 框架整合

## spring整合JDBC

### 步骤一：配置jdbc基本信息

<!-- 加载jdbc基本数据的属性文件-->

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<property name="locations">

<list>

<value>classpath:dbconfig.properties</value>

</list>

</property>

</bean>

<!-- jdbc直连数据库

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource"></bean>

-->

<!-- dbcp 数据连接池 -->

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName" value="${driverclass}"></property>

<property name="url" value="${url}"></property>

<property name="username" value="${username}"></property>

<property name="password" value="${password}"></property>

</bean>

<bean id="userDao" class="dao.UserDaoImpl">

<property name="jt" ref="jt"></property>----------------->在持久层中引入jdbc的模板类

</bean>

<!-- spring 提供一个jdbc的模版类 JdbcTemplate -->

<bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate">

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

</bean>

### 步骤二、创建持久层

1、让自己写的一个dao类继承JdbcDaoSupport

2、让自己写的一个dao类继承JdbcTemplate

3、让自己写的一个dao类里有一个属性为JdbcTemplate

public class UserDaoImpl extends JdbcDaoSupport implements UserDao {

public boolean add(User user) {

try{

String sql = "insert into user values(default,?,?)";

Object[] args = {user.getName(),user.getAge()};

int[] argTypes = {Types.VARCHAR,Types.INTEGER};

this.getJdbcTemplate().update(sql, args, argTypes);

} catch(Exception e) {

e.printStackTrace();

return false;

}

return true;

}

public User getById(int id) {

String sql = "select \* from user where id = ?";

Object[] objs = {id};

final User user = new User();

this.getJdbcTemplate().query(sql, objs, new RowCallbackHandler() {

public void processRow(ResultSet res) throws SQLException {

user.setId(res.getInt("id"));

user.setName(res.getString("name"));

user.setAge(res.getInt("age"));

}

});

return user;

}

public List<User> getByName2(String name) {

String sql = "select \* from user where name like ?";

Object[] objs = {"%"+name+"%"};

List<User> list = this.getJdbcTemplate().query(sql, objs, new RowMapper(){

public Object mapRow(ResultSet res, int i) throws SQLException {

User user = new User(res.getInt("id"),res.getString("name"),res.getInt("age"));

return user;

}

});

return list;

}

}

### 总结

1、引入dataSource的方式：

a. 在dataSource的设置中直接写值

b. 引入properties文件

2、在dao的写法中有很多种，最终只需要把dataSource注入到jdbcTemplate中

### 案例

public class PersonDao2 extends JdbcTemplate{

public PersonDao2(DataSource dataSource){

super(dataSource);

}

public void update(){

this.execute("update person set pname='aa' where pid=2");

}

}

public class PersonDao extends JdbcDaoSupport{

public void update(){

this.getJdbcTemplate().execute("update person set pname='a' where pid=3");

}

public void query(){

List<Person> persons = this.getJdbcTemplate().query("select \* from person", new PersonRowMapper());

for(Person person:persons){

System.out.println(person.getPname());

}

}

}

public class UserDaoImpl implements UserDao {

//引入jdbc的模板类

private JdbcTemplate jt;

public JdbcTemplate getJt() {

return jt;

}

public void setJt(JdbcTemplate jt) {

this.jt = jt;

}

@Override//增删改都是update();

public void insertUser(User user) {

String sql = "insert into user values(null,?,?)";//SQL语句

Object[] args = {user.getName(),user.getAge()};//将要经行DDL语句的操作的对象封装到object的数组中

int[] argTypes = {Types.VARCHAR,Types.INTEGER};//奖封装的属性的类型，封装到Int的数组中

jt.update(sql, args, argTypes);//开始执行

}

}

批量插入：需要引入一个需要导入的数据的集合，这个数据在传入的时候，必须是final，这样才能被内部类使用。

@Override

public void insertAllUser(final List<User> list) {

String sql = "insert into user values(null,?,?)";

jt.batchUpdate(sql, new BatchPreparedStatementSetter() {

@Override//i是循环的次数

public void setValues(PreparedStatement ps, int i) throws SQLException {

User user = list.get(i);

ps.setObject(1, user.getName());

ps.setObject(2, user.getAge());

}

@Override//list的长度，就是要插入数据的数量，也是这个方法内部的循环插入的次数

public int getBatchSize() {

return list.size();

}

});

}

回顾：

堆区的存活时间长，需要被垃圾回收的时候才被销毁

对象等new出来的时候，分配到内存的堆区

栈区的存活时间短，一般如方法，及其局部变量，在方法使用之后就被销毁。

对象的引用在栈区，

方法在栈区，

## Spring整合Hibernate

核心类：HibernateTemplate(spring)除了核心包之外还需要---spring的orm.jar,transaction.jar,jdbc.jar

### 把hibernate配置写在Spring的配置文件中

#### 步骤一：在applicationContext.xml中配置sessionFactory

<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

<property name="dataSource" ref="dataSource"></property>------->引入数据库的

<property name="hibernateProperties">

<props>

<prop key="hibernate.dialect">

org.hibernate.dialect.MySQLDialect

</prop>

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

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

</props>

</property>

<!-- 加载映射文件 ，一个一个的设置-->

<!-- <property name="mappingResources">

<list>

<value>entity/User.hbm.xml</value>

</list>

</property> -->

<!-- 加载映射文件 ，将一个包里的映射文件全部加载-->

<property name="mappingDirectoryLocations">

<list>

<value>classpath:entity</value>

</list>

</property>

</bean>

#### 步骤二：设置模板类

<bean id="ht" class="org.springframework.orm.hibernate3.HibernateTemplate">

<property name="sessionFactory" ref="sessionFactory"></property>---->向核心模板类中引入hibernate的全部数据

</bean>

#### 步骤三：设置持久层的超类

将核心模板类引入超类中，这样的话，持久层的类，就不需要每个都去设置了

<bean id="baseDao" class="com.project.dao.BaseDaoImpl" abstract="true">

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

</bean>

public class BaseDaoImpl<T, PK extends Serializable> implements BaseDao<T, PK> {

public Class entityClass;

public HibernateTemplate ht;

public HibernateTemplate getHt() {

return ht;

}

public void setHt(HibernateTemplate ht) {

this.ht = ht;

}

public boolean insert(T t) {

try{

ht.save(t);

} catch (Exception e) {

return false;

}

return true;

}

}

#### 步骤四：创建持久层

public class UserDaoImpl extends BaseDaoImpl<User, Integer> implements UserDao {

public UserDaoImpl() {

this.entityClass = User.class;

}

}

设置时要加属性---->parent="baseDao"指明它的父类，间接引入了核心模板类

<bean id="UserDao" class="com.project.dao.UserDaoImpl" parent="baseDao"></bean>

### 保留hibernate配置文件

<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

<property name="configLocation" value="classpath:hibernate.cfg.xml"></property>

</bean>

### 使用HQL语句

注意:导入hibernate是为了简化使用sql语句，所以，尽量使用HQL语句

#### 不使用session

public List<User> getByName(String name) {

String hql = "from User where name like '%"+name+"%'";

List<User> li = this.ht.find(hql);

return li;

}

#### 使用session---->建议使用，很多操作方便

public List<User> getByName2(final String name) {

List<User> li = ht.execute(new HibernateCallback() {

@Override

public Object doInHibernate(Session se) throws HibernateException, SQLException {

String hql = "from User where name like :name";

Query q = se.createQuery(hql);

q.setParameter("name", "%"+name+"%");

return q.list();

}

});

return li;

}

## ssh整合

struts2、spring、hibernate---->事务的管理交给了spring，更加的简化了代码，无需创建工具类

### 配置web.xml

#### struts2： 配置核心控制器

<filter>

<filter-name>struts</filter-name>

<filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>struts</filter-name>

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

</filter-mapping>

#### spring框架启动配置： listener

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/classes/applicationContext.xml</param-value>

</context-param>

#### 开启session

<filter>

<filter-name>opensession</filter-name>

<filter-class>org.springframework.orm.hibernate3.support.OpenSessionInViewFilter</filter-class>

<!-- 配置了这个过滤器以后,将不能进行增删改的操作 如何解决这个问题

1 添加事务控制

2 没有事务控制时，添加如下代码

-->

<init-param>

<param-name>singleSession</param-name>

<param-value>false</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>opensession</filter-name>

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

</filter-mapping>

#### spring提供的编码过滤器

<filter>

<filter-name>characterEncoding</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

</filter>

<filter-mapping>

<filter-name>characterEncoding</filter-name>

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

</filter-mapping>

### 配置applicationCo**n**text.xml

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver">

</property>

<property name="url" value="jdbc:mysql:///user"></property>

<property name="username" value="root"></property>

<property name="password" value="4100107223"></property>

</bean>

<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

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

<property name="hibernateProperties">

<props>

<prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>

</props>

</property>

<property name="mappingDirectoryLocations">

<list>

<value>classpath:entity</value>

</list>

</property>

</bean>

<!--1. 配置实体类 -->

<bean id="user" class="entity.User"></bean>

<bean id="emp" class="entity.Emp"></bean>

<bean id="dept" class="entity.Dept"></bean>

<!-- 向超类中添加sessionfactory -->

<bean id="baseDao" class="dao.BaseDaoImpl" abstract="true">

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

</bean>

<!-- 2.配置持久层 -->

<bean id="userDao" class="dao.UserDaoImpl" parent="baseDao"></bean>

<bean id="deptDao" class="dao.DeptDaoImpl" parent="baseDao"></bean>

<!-- 3.配置业务层 -->

<bean id="userService" class="service.UserServiceImpl">

<property name="userDao" ref="userDao"></property><!-- 想业务层中，注入需要的持久层 -->

</bean>

<bean id="deptService" class="service.DeptServiceImpl">

<property name="deptDao" ref="deptDao"></property><!-- 想业务层中，注入需要的持久层 -->

</bean>

<!-- 4.配置web层，作用是主要用来向action中注入持久层的对象 -->

<bean id="userAction" class="web.action.UserAction">

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

</bean>

<bean id="deptAction" class="web.action.DeptAction">

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

</bean>

<!-- 配置事务管理器 切面 -->

<bean id="txManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">

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

</bean>

<!-- 指定传播规则 -->

<tx:advice id="txAdvice" transaction-manager="txManager">

<tx:attributes> <!-- 事务传播规则 -->

<tx:method name="add\*" propagation="REQUIRED" isolation="REPEATABLE\_READ" />

<tx:method name="modify\*" propagation="REQUIRED"/>

<tx:method name="remove\*" propagation="REQUIRED"/>

<tx:method name="zhuanzhang" propagation="REQUIRED" no-rollback-for="java.lang.NullPointerException" />

<tx:method name="\*" propagation="REQUIRED"/>

</tx:attributes>

</tx:advice>

<!-- 哪些类的哪些方法 要被事务管理 -->

<aop:config>

<aop:pointcut id="allServiceMethod" expression="execution(\* service.\*.\*(..))"/>

<aop:advisor pointcut-ref="allServiceMethod" advice-ref="txAdvice" /><!-- 通知 -->

</aop:config>

# 回调

## js中的回调

指向函数的指针

xhr (XMLHttpRequest)

xhr.onreadystatechange=function(){

.........

}

你调用xhr的onreadystatechange xhr 回头调用你的实现

在这个对象的设计时，原 function onreadystatechange(){};方法体是空的，需要开发人员去编码，

执行的时候，是去调用你的实现。

## java中的回调

接口、匿名内部类

要调用某些方法的时候，它所在的类是一个接口，需要我们去实现这个接口，在执行这个方法的时候，它通过接口去调用你的代码

HibernateCallback(spring做的一个接口) ----- 回调接口

interface HibernateCallback{

public List doInHibernate();

}

class HibernateTemplate {--------Spring

execute(HibernateCallback hcb)

hcb.doInHibernate();

}

}

class Dao{ ----自己编写的类

HibernateTemplate ht

ht.execute(new HibernateCallback(){

public List doInHibernate(){

........

}

})

}

# Aop

aspect oriented pragramming 面向切编程，常见的AOP编程有：事务

## 可实现aop的功能的方法

Filter 过滤请求

Struts2的interceptor 只能拦截Action

广义的拦截器： 可以拦截任意 包括（POJO）

好处：

事务、日志、安全性框架、权限、目标方法之间完全是松耦合的

## aop的术语：

切面（Aspect）：拦截器,事务、日志、安全性框架、权限等都是切面

连接点（Joinpoint）：当切面 对某个切入点实行切入时，这个切入点就叫做连接点

通知（Advice）： 告诉切面以什么方式切目标（目标之前，目标之后，环绕，抛异常）

切入点（Pointcut）： 规定切面可以切目标的哪些位置，这些位置就是切入点

目标对象（Target）：要访问的对象

代理（Proxy）：目标和切面织在一起形成的新对象。对于客户对象来说，目标对象和代理对象是一样的

织入（Weaving）：动词 指的就是把目标和切面织在一起的动作

编译期

类装载期

运行期

## 在spring在，实现AOP编程的两种方式：

(1)aop联盟的方式

(2)aspectJ(现在主要使用这个)

### 步骤一：配置AOP

<beans

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

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

xmlns:p="http://www.springframework.org/schema/p"

xmlns:aop="http://www.springframework.org/schema/aop"------------->手动填入

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

<bean id="myAspect" class="com.zrgk.aop.MyAspect"></bean>--------->切入类

<bean id="myTarget" class="com.zrgk.aop.Target"></bean>----------->目标类

<aop:config>

<aop:aspect id="aspect" ref="myAspect">

<aop:pointcut expression="execution(\* com.zrgk.aop.\*.\*(..))" id="myPointcut" />----->要切的类，方法

<aop:before method="log" pointcut-ref="myPointcut" />------->使用那一个方法，去切

<aop:around method="testAround" pointcut-ref="myPointcut" />--------->使用那一个方法，去切

</aop:aspect>

</aop:config>

</beans>

注意：\*的代表一次是：返回类型，JavaBean，JavaBean中的方法

### 步骤二、创建AOP类

public class MyAspect {

public void log(JoinPoint jp){ //JoinPoint，可以获得当前目标类中的属性

System.out.println("aspect.......log.........");

System.out.println(jp.getSignature().getName());

}

public void testAround(ProceedingJoinPoint pjp){

System.out.println("around...before....");

try {

pjp.proceed(); //放行

} catch (Throwable e) {

e.printStackTrace();

}

System.out.println("around...after.....");

}

}

## 代理模式

### 静态代理模式的缺点：

1、如果一个系统中有100Dao，则创建100个代理对象

2、如果一个dao中有很多方法需要事务，则代理对象的方法中重复代码还是很多

3、由第一点和第二点可以得出：proxy的重用性不强

### 动态代理模式：

1、产生的代理对象和目标对象实现了共同的接口

jdk动态代理

2、代理对象是目标对象的子类

hibernate: Person person = session.load(Person.class,1L); javassisit

spring:cglib动态代理

3、jdk的动态代理：

1、因为是用jdk的API做到的

2、代理对象是动态产生的

cglib产生的代理类是目标类的子类

注意事项：

1、拦截器中invoke方法体的内容就是代理对象方法体的内容

2、当客户端执行代理对象.方法时，进入到了拦截器的invoke方法体

3、拦截器中invoke方法的method参数是在调用的时候赋值的

## 通知

1、前置通知

1、在目标方法执行之前执行

2、无论目标方法是否抛出异常，都执行，因为在执行前置通知的时候，目标方法还没有执行，还没有遇到异常

2、后置通知

1、在目标方法执行之后执行

2、当目标方法遇到异常，后置通知将不再执行

3、后置通知可以接受目标方法的返回值，但是必须注意：

后置通知的参数的名称和配置文件中returning="var"的值是一致的

3、最终通知：

1、在目标方法执行之后执行

2、无论目标方法是否抛出异常，都执行，因为相当于finally

4、异常通知

1、接受目标方法抛出的异常信息

2、步骤

在异常通知方法中有一个参数Throwable ex

在配置文件中

<aop:after-throwing method="throwingMethod" pointcut-ref="perform" throwing="ex"/>

5、环绕通知

1、如果不在环绕通知中调用ProceedingJoinPoint的proceed，目标方法不会执行

2、环绕通知可以控制目标方法的执行

# spring 声明式事务管理

事务的隔离级别、传播规则等详看MySQL

## Java代码分层的作用

是为了提高代码的复用性

持久层的方法，要做到原子化，使dao中每个方法，都只做一件事。

业务层 ，组合持久层的方法，完成一个业务，也就是一个完整的事务。

## 数据库事务是

将一组的sql单元，完成一个工作。

## Spring事务使用规范

如果代码中加了事务，必须做好开发环境测试（测试环境也尽量触发异常、测试回滚），确保事务生效。以下列了事务使用过程的注意事项，请大家留意。

1.不要在接口上声明@Transactional ，而要在具体类的方法上使用 @Transactional 注解，否则注解可能无效。

2.不要图省事，将@Transactional放置在类级的声明中，放在类声明，会使得所有方法都有事务。故@Transactional应该放在方法级别，不需要使用事务的方法，就不要放置事务，比如查询方法。否则对性能是有影响的。

3.使用了@Transactional的方法，对同一个类里面的方法调用， @Transactional无效。比如有一个类Test，它的一个方法A，A再调用Test本类的方法B（不管B是否public还是private），但A没有声明注解事务，而B有。则外部调用A之后，B的事务是不会起作用的。（经常在这里出错）

4.使用了@Transactional的方法，只能是public，@Transactional注解的方法都是被外部其他类调用才有效，故只能是public。道理和上面的有关联。故在 protected、private 或者 default 的方法上使用 @Transactional 注解，它也不会报错，但事务无效。

经过测试，效果如下：

A.抛出运行时异常比如NullPointerException、，事务会回滚。

B.Quartz中，execute直接调用加了@Transactional方法，可以回滚；间接调用，不会回滚。（即上文3点提到的）

C.异步任务中，execute直接调用加了@Transactional方法，可以回滚；间接调用，不会回滚。（即上文3点提到的）

D.在action中加上@Transactional，不会回滚。切记不要在action中加上事务。

E.在service中加上@Transactional，如果是action直接调该方法，会回滚，如果是间接调，不会回滚。（即上文3提到的）

F.在service中的只有public的方法加上@Transactional，事务会回滚。

## 在SSH整合框架里使用事务

### 步骤一、配置

<!-- 配置事务管理器，切面 -->

<bean id="txManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">

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

</bean>

<!-- 指定传播规则 -->

<tx:advice id="txAdvice" transaction-manager="txManager">

<tx:attributes> <!-- 方法 事务传播规则 事务隔离级别 -->

<tx:method name="add\*" propagation="REQUIRED" isolation="REPEATABLE\_READ" />

<tx:method name="modify\*" propagation="REQUIRED"/>

<tx:method name="remove\*" propagation="REQUIRED"/> 出现那些异常时，不回滚

<tx:method name="zhuanzhang" propagation="REQUIRED" no-rollback-for="java.lang.NullPointerException" />

<tx:method name="\*" propagation="REQUIRED"/>

</tx:attributes>

</tx:advice>

<!-- 哪些类的哪些方法 要被事务管理 -->

<aop:config>

<aop:pointcut id="allServiceMethod" expression="execution(\* service.\*.\*(..))"/>

<aop:advisor pointcut-ref="allServiceMethod" advice-ref="txAdvice" /><!-- 通知 -->

</aop:config>

# Spring注解

## 初始配置

spring要使用注解方式开发，要启动spring的注解方法

在spring的配置文件中启动：

启动spring的注解：

<context:annotation-config></context:annotation-config>

设置spring扫描bean：

<context:component-scan base-package="com.zrgk"></context:component-scan>

--spring将会扫描com.zrgk 及其子包下的 所有添加了 @Component注解的 类 当成spring bean

## 常用注解

@Component 加在类名上 表示这个类会被解析成spring的bean

当不给这个类指定bean的名字时，默认采用类名首字母变小写的方式

也可以手动指定@Component("bean的名字") @Component(value="bean的名字")

@Autowired 可以加在属性 和set方法上

自动装配 先按类型装配 当有多个复合的bean时 如何装配 ---不能装配 报错

@Qualifier @Qualifier("要被注入的bean的名字") 要结合 @Autowired 就避免了上面的问题

@Scope("prototype")解决bean单例问题

## web工程中的spring注解

@Repository、@Service和@Controller 这三个注解的作用和 @Component 相同 用来把类指定为bean

@Repository 加在 持久层

@Service 加在 业务层

@Controller 加在 控制层--->web层

@Component 一般都是加载实体类中，工具，切面等

这三个注解和 @Component 有和区别 ？

目前的spring中 没有任何区别

为以后的版本做准备

## spring AOP的注解方式：

想要开启aspectJ的注解方式，必须启动spring的注解

<aop:aspectj-autoproxy/>

1.在切面类上添加： @Aspect

2.定义一个任意方法，为什么要定义一个任意方法？？？ 因为@Pointcut 要求要放在一个方法

@Component

@Aspect

public class MyAspect {

@Pointcut("execution(\* com.zrgk.aop.\*.\*(..))")

public void myMethod() {

}

@Before("myMethod()")

public void before(JoinPoint jp) {

System.out.println("myaspect...before.....");

System.out.println(jp.getSignature().getName());

}

@After("myMethod()")

public void after() {

System.out.println("myaspect...after.....");

}

@Around("anyMethod()")

public void aroundTest(ProceedingJoinPoint pjp){

System.out.println("around...before....");

try {

pjp.proceed();

} catch (Throwable e) {

e.printStackTrace();

}

System.out.println("around...after....");

}

}

3 @Before @After @Around @AfterReturning @AfterThrowing

4 用法：

@Pointcut("execution(\* com.\*.\*(..))")

public void anyMethod(){

}

@Before("anyMethod()")

public void log(JoinPoint){

System.out.println("myAspect....log....before");

}

@Around("anyMethod()")

public void aroundTest(ProceedingJoinPoint pjp){

System.out.println("around...before....");

try {

pjp.proceed();

} catch (Throwable e) {

e.printStackTrace();

}

System.out.println("around...after....");

}

## ssh 注解方式整合

### 将hibernate.cfg.xml，配置到applicationContext.xml中

配置sessionFactory

LocaleSessionFactory---->AnnotationSessionFactoryBean

BaseDaoImpl 也不能在 spring的配置文件中进行配置，需要在BaseDaoImpl注入，必须继承HibernateDaoSupport

@Autowired

public void addSessionFactory(SessionFactory sessionFactory){

super.setSessionFactory(sessionFactory);

}

### 注解方式的 事务控制

注解方式的事务控制 比在配置文件中使用aspectJ方式配置 要麻烦-- 所以注解方式的事务控制实际上并不好，因为需要事务控制的类上都需要添加注解

1. 开启aspectJ的注解方式

2 . 也要配置事务管理器

<bean id="transactionManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">

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

</bean>

3 指定注解方式时使用的事务管理器

<tx:annotation-driven transaction-manager="transactionManager" proxy-target-class="true" />

4 @Transactional

1 可以加在类上 ---> 对类中所有public 的方法进行事务控制

1，对本类中的公共方法有效。

2，对子类中的公共方法有效（即这个注解可以被继承）

3，对父类中的方法无效！！！

2 也可以加在具体方法上

## @Resource注解的使用规则：

1、在spring的配置文件中导入命名空间

xmlns:context="http://www.springframework.org/schema/context"

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-2.5.xsd

2、引入注解解析器

<context:annotation-config></context:annotation-config>

3、在spring的配置文件中把bean引入进来

4、在一个类的属性上加

@Resource(name="student\_annotation")

private Student student;

从该注解本身

@Target({TYPE, FIELD, METHOD})

@Retention(RUNTIME)

public @interface Resource {

String name() default "";

}

1、该注解可以用于属性上或者方法上，但是一般用户属性上

2、该注解有一个属性name,默认值为""

5、分析整个过程：

1、当启动spring容器的时候，spring容器加载了配置文件

2、在spring配置文件中，只要遇到bean的配置，就会为该bean创建对象

3、在纳入spring容器的范围内查找所有的bean,看哪些bean的属性或者方法上加有@Resource

4、找到@Resource注解以后，判断该注解name的属性是否为""(name没有写)

如果没有写name属性，则会让属性的名称的值和spring中ID的值做匹配，如果匹配成功则赋值

如果匹配不成功，则会按照类型进行匹配，如果匹配不成功，则报错

如果有name属性，则会按照name属性的值和spring的bean中ID进行匹配，匹配成功，则赋值，不成功则报错

## 类扫描的注解

1、在spring的配置文件中导入命名空间

xmlns:context="http://www.springframework.org/schema/context"

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-2.5.xsd

2、<context:component-scan base-package="cn.itcast.annotation.scan"></context:component-scan>

1、 该注解解析器包含了两个功能：依赖注入和类扫描

2、在base-package包及子包下查找所有的类

3、如果一个类上加了@Component注解，就会进行如下的法则

如果其value属性的值为""

@Component

public class Person {}

==

<bean id="person" class="..Person">

如果其value属性的值不为""

@Component("p")

public class Person {}

==

<bean id="p" class="..Person">

4、按照@Resource的法则再次进行操作

依赖注入

@Resource

@Autowired

@Qualifier

类扫描

@Component

@Controller

@Repository

@Service

# 好用的工具类

下面提到工具类都是在org.springframework.util下面

## 内置的resouce类型

UrlResource

ClassPathResource

FileSystemResource

ServletContextResource

InputStreamResource

ByteArrayResource

VfsResource(在jboss里经常用到, 相应还有 工具类 VfsUtils)

org.springframework.util.xml.ResourceUtils 用于处理表达资源字符串前缀描述资源的工具. 如: &quot;classpath:&quot;.

有 getURL, getFile, isFileURL, isJarURL, extractJarFileURL

## 工具类

org.springframework.core.annotation.AnnotationUtils   处理注解

org.springframework.core.io.support.PathMatchingResourcePatternResolver  用于处理 ant 匹配风格(com/\*.jsp, com/\*\*/\*.jsp),找出所有的资源, 结合上面的resource的概念一起使用,对于遍历文件很有用. 具体请详细查看javadoc

org.springframework.core.io.support.PropertiesLoaderUtils 加载Properties资源工具类,和Resource结合

org.springframework.core.BridgeMethodResolver  桥接方法分析器.  关于桥接方法请参考: http://java.sun.com/docs/books/jls/third\_edition/html/expressions.html#15.12.4.5

org.springframework.core.GenericTypeResolver  范型分析器, 在用于对范型方法, 参数分析.

org.springframework.core.NestedExceptionUtils

## xml工具

org.springframework.util.xml.AbstractStaxContentHandler

org.springframework.util.xml.AbstractStaxXMLReader

org.springframework.util.xml.AbstractXMLReader

org.springframework.util.xml.AbstractXMLStreamReader

org.springframework.util.xml.DomUtils

org.springframework.util.xml.SimpleNamespaceContext

org.springframework.util.xml.SimpleSaxErrorHandler

org.springframework.util.xml.SimpleTransformErrorListener

org.springframework.util.xml.StaxUtils

org.springframework.util.xml.TransformerUtils

## 和web相关的工具

org.springframework.web.util.CookieGenerator

org.springframework.web.util.HtmlCharacterEntityDecoder

org.springframework.web.util.HtmlCharacterEntityReferences

org.springframework.web.util.HtmlUtils

org.springframework.web.util.HttpUrlTemplate

这个类用于用字符串模板构建url, 它会自动处理url里的汉字及其它相关的编码. 在读取别人提供的url资源时, 应该经常用

String url = &quot;http://localhost/myapp/{name}/{id}&quot;

org.springframework.web.util.JavaScriptUtils

org.springframework.web.util.

org.springframework.web.util.WebUtils

org.springframework.web.util.UriUtils处理uri里特殊字符的编码

org.springframework.web.util.UriTemplate

用listener的方式来配制log4j在web环境下的初始化

## 其它工具集

org.springframework.util.xml.AntPathMatcherant风格的处理

org.springframework.util.xml.AntPathStringMatcher

org.springframework.util.xml.Assert断言,在我们的参数判断时应该经常用

org.springframework.util.xml.CachingMapDecorator

org.springframework.util.xml.ClassUtils用于Class的处理

org.springframework.util.xml.CollectionUtils用于处理集合的工具

org.springframework.util.xml.CommonsLogWriter

org.springframework.util.xml.CompositeIterator

org.springframework.util.xml.ConcurrencyThrottleSupport

org.springframework.util.xml.CustomizableThreadCreator

org.springframework.util.xml.DefaultPropertiesPersister

org.springframework.util.xml.DigestUtils摘要处理, 这里有用于md5处理信息的

org.springframework.util.xml.FileCopyUtils文件的拷贝处理, 结合Resource的概念一起来处理, 真的是很方便

org.springframework.util.xml.FileSystemUtils

org.springframework.util.xml.LinkedCaseInsensitiveMap

key值不区分大小写的LinkedMap

org.springframework.util.xml.LinkedMultiValueMap一个key可以存放多个值的LinkedMap

org.springframework.util.xml.Log4jConfigurer一个log4j的启动加载指定配制文件的工具类

org.springframework.util.xml.NumberUtils处理数字的工具类, 有parseNumber 可以把字符串处理成我们指定的数字格式, 还支持format格式, convertNumberToTargetClass 可以实现Number类型的转化.

org.springframework.util.xml.ObjectUtils有很多处理null object的方法. 如nullSafeHashCode, nullSafeEquals, isArray, containsElement, addObjectToArray, 等有用的方法

org.springframework.util.xml.PatternMatchUtilsspring里用于处理简单的匹配. 如 Spring's typical &quot;xxx\*&quot;, &quot;\*xxx&quot; and &quot;\*xxx\*&quot; pattern styles

org.springframework.util.xml.PropertyPlaceholderHelper用于处理占位符的替换

org.springframework.util.xml.ReflectionUtils反映常用工具方法. 有 findField, setField, getField, findMethod, invokeMethod等有用的方法

org.springframework.util.xml.SerializationUtils用于java的序列化与反序列化. serialize与deserialize方法

org.springframework.util.xml.StopWatch一个很好的用于记录执行时间的工具类, 且可以用于任务分阶段的测试时间. 最后支持一个很好看的打印格式. 这个类应该经常用

org.springframework.util.xml.StringUtils

org.springframework.util.xml.SystemPropertyUtils

org.springframework.util.xml.TypeUtils用于类型相容的判断. isAssignable

org.springframework.util.xml.WeakReferenceMonitor弱引用的监控

org.springframework.web.bind.ServletRequestUtils

//取请求参数的整数值：

public static Integer getIntParameter(ServletRequest request, String name)

public static int getIntParameter(ServletRequest request, String name, int defaultVal) -->单个值

public static int[] getIntParameters(ServletRequest request, String name) -->数组

还有譬如long、float、double、boolean、String的相关处理方法。

对象序列化与反序列化 org.springframework.util.SerializationUtils

public static byte[] serialize(Object object)

public static Object deserialize(byte[] bytes)

数字处理 org.springframework.util.NumberUtils

字符串转换为Number并格式化，包括具体的Number实现类，如Long、Integer、Double，字符串支持16进制字符串，并且会自动去除字符串中的空格：

public static <T extends Number> T parseNumber(String text, Class<T> targetClass)

public static <T extends Number> T parseNumber(String text, Class<T> targetClass, NumberFormatnumberFormat)

各种Number中的转换，如Long专为Integer，自动处理数字溢出（抛出异常）：

public static <T extends Number> T convertNumberToTargetClass(Number number, Class<T> targetClass)

文件复制 org.springframework.util.FileCopyUtils

流与流之间、流到字符串、字节数组到流等的复制

目录复制 org.springframework.util.FileSystemUtils

只有两个方法，递归复制和递归删除一个目录

MD5加密 org.springframework.util.DigestUtils

字节数组的MD5加密 public static String md5DigestAsHex(byte[] bytes)

# Valid参数校验

<https://mp.weixin.qq.com/s/Mo3Mff789zAVvS91Ub6OnQ>

校验放在哪？

Controller层 VS Service层

去网上查阅了一些资料，一般推荐与业务无关的放在Controller层中进行校验，而与业务有关的放在Service层中进行校验。

使用Hibernate Validate

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-validator</artifactId>

<version>4.3.1.Final</version>

</dependency>

常用案例：

public class DataSetSaveVO {

//唯一标识符为空

@NotBlank(message = "user uuid is empty")

//用户名称只能是字母和数字

@Pattern(regexp = "^[a-z0-9]+$", message = "user names can only be alphabetic and numeric")

@Length(max = 48, message = "user uuid length over 48 byte")

private String userUuid;

//数据集名称只能是字母和数字

@Pattern(regexp = "^[A-Za-z0-9]+$", message = "data set names can only be letters and Numbers")

//文件名称过长

@Length(max = 48, message = "file name too long")

//文件名称为空

@NotBlank(message = "file name is empty")

private String name;

//数据集描述最多为256字节

@Length(max = 256, message = "data set description length over 256 byte")

//数据集描述为空

@NotBlank(message = "data set description is null")

private String description;

}

Controller写法

@PostMapping

public ResponseVO createDataSet(@Valid @RequestBody DataSetSaveVO dataSetVO) {

return ResponseUtil.success(dataSetService.saveDataSet(dataSetVO));

}