Spring的两大核心：IOC和AOP

解耦的思路：

第一步。使用反射来创建对象，而避免new关键字

第二步。通过读取配置文件来获取要创建的对象的全限定类名

Resources文件夹里创建一个xxxx.xml

<?xml version="1.0" encoding="UTF-8"?>

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

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

xsi:schemaLocation="http://www.springframework.org/schema/beans

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

将需要Spring管理的类写在<bean>中

<!--把对象的创建交给spring管理-->  
<bean id="accountService" class="com.liwenshuo.service.impl.AccountServiceImpl"></bean>

此段代码会创建并返回一个AccountServiceImpl对象

ApplicationContext的三个常用实现类

\* ClassPathXmlApplicationContext,它可以加载类路径下的配置文件，要求配置文件必须在路径下，不在的话，加载不了

\* FileSystemXmlApplicationContext,它可以加载磁盘任意路径下的配置文件

\* AnnotationConfigApplicationContext,它是用于读取注解创建容器的

核心容器的两个接口引发出的问题：

\* ApplicationContext： 单例对象适用

\* 它在构建核心容器时，创建对象采取的策略是立即加载的方式。也就是说，只要一读取完配置文件马上就创建配置文件中配置的对象

\* BeanFactory: 多例对象适用

\* 它在构建核心容器时，创建对象采取的策略是延迟加载的方式。也就是说什么时候根据id获取对象了，什么时候才真正创建对象

在表现层里，获取容器里的对象

//1.获取核心容器

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

//ApplicationContext ac = new FileSystemXmlApplicationContext("D:\\workspace\\springdemo\\SpringIOC\\src\\main\\resources\\bean.xml");

//2.根据id获取Bean对象

IAccountService as = (IAccountService) ac.getBean("accountService");

IAccountDao adao = (IAccountDao) ac.getBean("accountDao",IAccountDao.class);

spring对bean的管理细节

1.创建bean的三种方式

2.bean对象的作用范围

3.bean对象的生命周期

创建bean的三种方式

第一种方式：使用默认构造函数创建

在spring的配置文件中使用bean标签，配以id和class属性之后，且没有其他属性和标签时

采用的就是默认构造函数创建bean对象，此时如果类中没有默认构造函数，对象无法被创建

<bean id="accountService" class="com.liwenshuo.service.impl.AccountServiceImpl"></bean>

第二种方式：使用普通工厂的方法来创建对象（使用某个类中的方法创建对象，并存入spring容器）

<bean id="instanceFactory" class="com.liwenshuo.factory.InstanceFactory"></bean>

<bean id="accountService" factory-bean="instanceFactory" factory-method="getAccountService"></bean>

第三种方式：使用工厂中的静态方法创建对象（使用某个类中的静态方法创建对象，并存入spring容器）

<bean id="accountService" class="com.liwenshuo.factory.StaticFactory" factory-method="getAccountService"></bean>

bean的作用范围

bean标签的scope属性：

作用：用于指定bean的作用范围

取值： singleton:单例

prototype：多例

request：web请求范围

session：web应用的会话范围

global-session：用于集群环境（全局会话范围），当不是集群环境时，他就是session

多例：<bean id="accountService" class="com.liwenshuo.service.impl.AccountServiceImpl" scope="prototype" ></bean>

集群环境：（global-session的环境）

已登录为例：比如有6个服务器，当第一次请求时，A服务器空闲，A服务器返回给客户端一个验证码，当填完验证码第二次向服务器提交请求时，A服务器满了，B服务器空闲，第二次请求就到了B服务器，但是验证码存在了A服务器的cookie中，B不知道，这样会登录失败。然而把6个服务器看成一个服务器，那么存在A服务器中的验证码，同时存放在6个服务器中，这6个服务器就组成一个集群环境。

bean对象的生命周期

单例singleton:

出生：当容器创建时

活着：只要容器还在，对象就在

死亡：容器销毁，对象消亡

多例prototype:

出生：当使用对象时，spring框架为我们创建

活着：对象在使用过程一直活着

死亡：当对象长时间不用，且没有别的对象引用时，由java的垃圾回收器回收

依赖注入

依赖关系管理：

以后都交给spring维护

在当前类需要用到其它类的对象，由spring为我们提供，我们只需要在配置文件中说明

依赖关系的维护：

就称之为依赖注入

依赖注入：

能注入的数据，3类

基本类型和String

其他Bean类型（在配置文件中或者注解配置过的bean）

复杂类型/集合类型

注入的方式，3种

构造函数提供

使用set方法提供

使用注解提供

构造方法注入（全参构造）

使用标签：constructor-arg

标签的位置：bean标签的内部

标签中的属性

type:用于指定要注入的数据的数据类型，该数据类型也是构造函数中的某个或某些参数的类型

index：用于指定要注入的数据给构造函数中的指定索引位置的参数赋值，索引的位置是从0开始

常用 name：用于指定给构造函数中指定名称的参数赋值

======以上三个用于指定构造函数中的哪个参数赋值==========

value：用于提供基本类型和String类型的数据

ref：用于指定其他的bean类型数据，它指的就是在spring的IOC核心容器中出现过的bean对象

优势：在获取对象时，注入数据是必须的操作，否则对象无法创建成功

缺点：改变了bean对象的实例化方式，使我们在创建对象时，即使用不到这些数据，也必须提供

<bean id="accountService" class="com.liwenshuo.service.impl.AccountServiceImpl" >

<constructor-arg name="name" value="张三"></constructor-arg>

<constructor-arg name="age" value="20"></constructor-arg>

<constructor-arg name="birthday" ref="now"></constructor-arg>

</bean>

当数据类型不是基本数据类型和String类型时，需要使用ref，不再使用value，并且需要配置对象，比如date日期类型 <bean id="now" class="java.util.Date"></bean>

使用set方法注入

涉及的标签：property

出现的位置：bean标签的内部

标签中的属性

name：用于指定给构造函数中指定名称的参数赋值只关心set方法的名称

value：用于提供基本类型和String类型的数据

ref：用于指定其他的bean类型数据，它指的就是在spring的IOC核心容器中出现过的bean对象

优势：创建对象时没有明确的限制，可以直接使用默认构造方法

缺点：如果某个成员必须有值，则获取对象时有可能set方法没有执行

<bean id="accountService2" class="com.liwenshuo.service.impl.AccountServiceImpl2">

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

<property name="age" value="23"></property>

<property name="birthday" ref="now"></property>

</bean>

集合类型的注入：

Properties，map，通用

<property name="myMap">

<map>

<entry key="testa" value="AAA"></entry>

<entry key="tsetb" value="BBB"></entry>

</map>

</property>

数组，list，set，通用

<property name="mylist">

<list>

<value>AAA</value>

<value>BBB</value>

</list>

</property>

注解简单分类：

\* 用于创建对象的

\* 他们的作用和在xml配置文件中编写一个<bean>实现的功能是一样的

\* @Component: 起别名@Component(value="accountService")=@Component("accountService")

\* 作用：用于把当前类对象存入spring容器中

\* 属性：

\* value：用于指定bean的id，当我们不写时，它的默认值是当前类名，且首字母改小写

\* @Controller：一般用于表现层

\* @Service：一般用在业务层

\* @Repository：一般用在持久层

\* 以上三个注解他们的作用和属性与@Component是一摸一样的。他们三个是spring框架为我们提供明确的三层使用注解，是我们三层更加清晰

\* 用于注入数据的

\* 他们的作用和在xml配置文件中编写一个<properties>实现的功能是一样的

\* @Autowired:

\* 作用：自动按照类型注入。只要容器中有唯一的一个bean对象类型和要注入的变量类型匹配，就可以注入成功

\* 如果ioc容器中没有任何bean的类型和要注入的变量类型匹配，则报错。

\* 如果ioc容器中有多个类型匹配时，会报错

\* 出现位置：

\* 可以是变量上，也可以是方法上

\* 细节：

\* 在使用注解注入时，set方法就不是必须的了。

\* @Qualifier:

\* 作用：在按照类型注入的基础上再按照名称注入。它在给类成员注入时不能单独使用。但是在方法参数注入时可以

\* 属性：value：用于指定注入的id

\* @Resource

\* 作用：直接按照bean的id注入。他可以单独使用

\* 属性：

\* name：用于指定bean的id。

\* 以上三个注解都只能注入其他bean类型的数据，而基本类型和String类型无法使用上述注解实现。

\* 另外，集合类型的注入只能通过xml文件来实现

\* @Value：

\* 用于注入基本数据类型和String类型

\* 属性：

\* value：用于指定数据的值。它可以使用spring中SpEL（也就是spring的el表达式）

\* SpEL的写法，${表达式}

\* 用于改变作用范围的

\* 他们的作用和在bean标签中的scope属性实现的功能是一样的

\* @Scope

\* 作用：用于指定bean的作用范围

\* 属性：

\* value：指定范围的取值。常用取值：singleton，prototype

\* 和生命周期相关的 了解

\* 他们的作用和在bean标签中的init-method，destroy-method属性实现的功能是一样的

\* PreDestroy：用于指定销毁方法

\* PostConstruct:用于指定初始化

数据库Connection涉及到线程不安全的问题。

/\*连接的工具类，它用于从数据源中获取一个连接，并且实现和线程的绑定\*/

public class ConnectionUtils {

private ThreadLocal<Connection> t1 = new ThreadLocal<Connection>();

private DataSource dataSource;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

}

//获取当前线程上的连接

public Connection getConnection(){

try {

//1.先从ThreadLocal上获取

Connection conn = t1.get();

//2.判断当前线程上是否有连接

if (conn == null) {

//3.从数据源中获取一个连接，并且存入ThreadLocal中

conn = dataSource.getConnection();

t1.set(conn);

}

//4.返回当前线程上的连接

return conn;

}catch (Exception e ){

throw new RuntimeException(e);

}

}

//把线程和连接断开

public void removeConnection(){

t1.remove();

}

}

比如一些涉及到两个账户的交互功能，比如银行的转账功能

public class TransactionManager {

private ConnectionUtils connectionUtils;

public void setConnectionUtils(ConnectionUtils connectionUtils) {

this.connectionUtils = connectionUtils;

}

//开启事务

public void begin() {

try {

connectionUtils.getConnection().setAutoCommit(false);

} catch (Exception e) {

e.printStackTrace();

}

}

//提交事务

public void commit() {

try {

connectionUtils.getConnection().commit();

} catch (Exception e) {

e.printStackTrace();

}

}

//回滚事务

public void rollback() {

try {

connectionUtils.getConnection().rollback();

} catch (Exception e) {

e.printStackTrace();

}

}

//释放连接

public void release(){

try {

connectionUtils.getConnection().close(); //还回连接池中

connectionUtils.removeConnection(); //解绑，将线程和连接解绑

} catch (Exception e) {

e.printStackTrace();

}

}

}

动态代理

AOP

基于动态代理

相关术语：Joinpoint连接点：这些点指的是方法，因为spring只支持方法类型的连 接点

Pointcut切入点：指的是我们要对哪些joinpoint进行拦截的定义

织入：

通知：整个方法就是环绕通知

目标代码

之前的就是前置通知

之后就是后置通知

异常里面的是异常通知

Finally里面的就是最终通知

Spring中的JdbcTemplate

作用：和数据库交互的，实现对表的CRUD操作

如何创建该对象：

对象中的常用方法：

工厂注入：

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

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

</bean>

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

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

<property name="url" value="jdbc:mysql://localhost:3306/test"></property>

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

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

</bean>

CRUD操作，以下写法是测试人员用的

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

JdbcTemplate jt = (JdbcTemplate) ac.getBean("jdbcTemplate");

//保存

jt.update("insert into account(name,money) values (?,?)","aaa",55f);

//更新

jt.update("update account set name =?,money = ? where id = ?" ,"bbb",100f,3);

//删除

jt.update("delete from account where id = ?",4);

//查询所有

List<Account> list = jt.query("select \* from account where money>?", new BeanPropertyRowMapper<Account>(Account.class), 50f);

for (Account account:list) {

System.out.println(account);

}

//查询一个

List<Account> list1 = jt.query("select \* from account where id =?", new BeanPropertyRowMapper<Account>(Account.class), 5);

System.out.println(list1.isEmpty()?"没有内容":list1.get(0));

//返回一行一列

Long account = jt.queryForObject("select count(\*) from account where money > ?", Long.class, 50f);

System.out.println(account);

作为开发人员：需要编写dao层接口