# 1.spring入门

## 1.1.什么是框架:

框架就是将一些通用的重要的模块进行整合,直接给调用者使用即可.程序员无需关注通用模块的具体步骤.

（我认为框架就是一系列的底层服务，提供给我们开发者直接调用来快速开发软件）

举个简单的例子：我们之前的easymall，不同层之间调用，需要些一个工厂类，事务，要写一个事务类，和数据库连接，要写一大堆jdbc代码，甚至是jsp往servlet传值，要写一大堆获取代码，非常繁琐。而学完框架以后，只需要一些xml配置，一切都搞定了，再也不用那么繁琐。其实，我们学习框架，主要就是学习如何配置我们的xml配置。

SSH:

Struts2:主要是用户通过浏览器与程序进行交互(安全性不好,配置特别的繁琐)

Spring框架:整合现阶段所有的主流框架,能够让程序调用浑然一体.(粘合剂)

Hibernate:主要与数据库进行交互(特点:产生非常多的冗余的sql,如果单表数据量超过500万时,查询速度较慢)

SSM:

SpringMVC

Spring

Mybatis

## 1.2.课程介绍:

2.1.Spring的IOC和DI 重点知识

2.2.Spring中的简化配置

2.3.Spring的AOP

2.4.Spring中AOP的重点练习

2.5.Spring整合JDBC,与数据库事务的控制

## 1.3.Spring的IOC

IOC:控制反转

说明:就是将对象创建的权利交给Spring容器来完成,从此程序员无需关注对象的生命周期.

以前:Person p1 = new Person(); 自己手动创建的

现在:Person p2 = spring容器.getXXX();

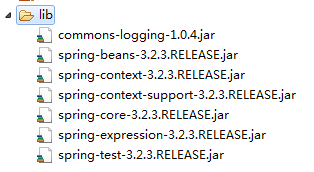
接下来，我们来看一个spring的入门案例，今天我们说过，我们学习框架，主要就是学xml配置的，但你们也无需担心，现在可能我会写很多的配置，但是到最后，可能一两行就搞定了，所以这两天你们会学的比较轻松，但是重要的是思想。

## 1.4.IOC的入门案例

步骤:

4.1.导入spring的jar包7个

建一个java工程，新建一个lib文件夹，在课前资料中导入jar包，并build path



4.2.编写实体类

|  |
| --- |
| package pojo;  public class Hello {  public void sayHello(){  System.out.println("Hello Spring");  }  } |

4.3.编写配置文件(1.导入配置文件的头 ,2编写实体内容)

4.3.1.在src下创建applicationContext.xml文件（spring默认名字，行业通用）

4.3.2.导入配置文件的头

4.3.3.编写实体内容

|  |
| --- |
| <?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-3.2.xsd ">  <!-- bean:只要被spring管理的类就是bean  id：bean的唯一标识(通常就是类名首字母小写）  class：被spring管理的类  -->  <bean id="hello" class="pojo.Hello"></bean>  </beans> |

4.4.启动spring容器，获取对象，完成操作

|  |
| --- |
| package test;  import org.junit.Test;  import org.springframework.context.ApplicationContext;  import org.springframework.context.support.ClassPathXmlApplicationContext;  import pojo.Hello;  public class TestSpring {  @Test  public void test01(){  //启动spring容器  ApplicationContext context =  new ClassPathXmlApplicationContext("applicationContext.xml");    //获取对象：通过bean的id, 通过bean获取的对象为object，需要强转  Hello hello = (Hello) context.getBean("hello");  hello.sayHello();  }  } |

注意：当spring容器启动的时候，会先加载spring的核心配置文件applicationContext.xml，逐行解析，当解析到bean标签时，根据class通过反射为其创建对象，与id一同保存到spring维护的map中，key为id，value为创建的对象。

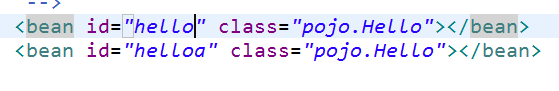
4.5.spring容器获取对象的方式：（偏底层，重要的是思想）

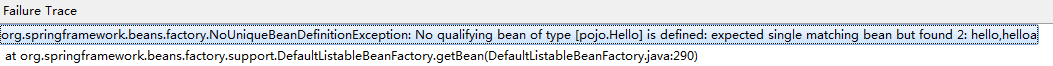
4.5.1.通过bean中的id获取

4.5.2.通过class类型获取。

|  |
| --- |
| //获取对象：通过class类型与bean中的class做匹配，如果匹配上，则从bean中的id从容器中获取对象。  Hello hello = (Hello) context.getBean(Hello.class); |

如果配置多个相同class的bean，则报错





建议：通过id获取，因为通过类型可能会造成多个id报错。

4.6.别名标签（比如软件升级的时候，为了区分，你将之前的id要换一个，但是如果直接换了，之前的代码就不能用了，所以就可以使用别名标签）

在applicationContext.xml中添加别名标签

|  |
| --- |
| <!-- 添加别名标签，name为需要添加别名的bean的id，alias为别名 -->  <alias name="hello" alias="hi"/> |

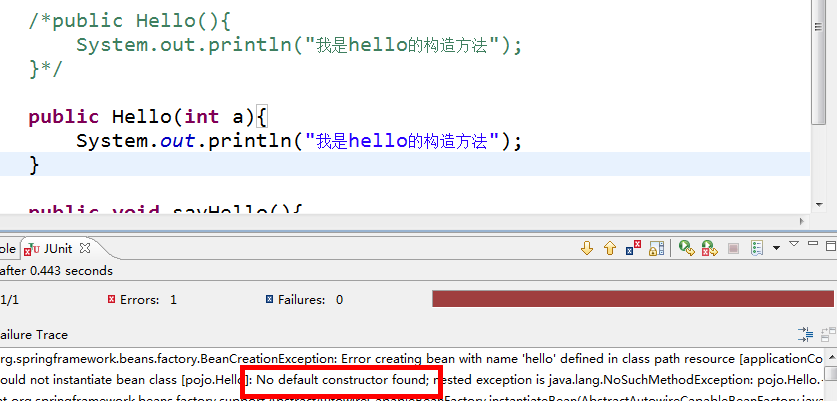
测试类测试：

|  |
| --- |
| //测试通过别名标签获取对象  @Test  public void test03(){  //启动spring容器  ApplicationContext context =  new ClassPathXmlApplicationContext("applicationContext.xml");  //获取对象：通过别名获取  Hello hello = (Hello) context.getBean("hi");  hello.sayHello();  } |

4.7.spring容器创建对象的方式

4.7.1.通过默认构造方法创建（通过反射创建对象的方式就是这个）

反射需要依赖默认构造方法创建对象，所以，以后在写构造方法的时候要加上无参构造，这样才能正常创建



4.7.2.如果遇到不能直接new的对象该如何处理，比如接口，抽象类

通过静态工厂创建对象：

说明：必须在方法上添加static关键字，可以通过class直接调用

步骤：首先，创建静态工厂类

|  |
| --- |
| package factory;  import java.util.Calendar;  public class StaticFactoty {  public StaticFactoty(){  System.out.println("我是静态工厂的构造方法");  }  public static Calendar getCalendar(){  return Calendar.getInstance();  }  } |

applicationContext.xml中配置文件

|  |
| --- |
| <!-- 通过静态工厂获取对象 class:静态工厂的class  factory-method：需要创建对象的方法-->  <bean id="calendar" class="factory.StaticFactoty"  factory-method="getCalendar"></bean> |

测试

|  |
| --- |
| //测试静态工厂  @Test  public void test04(){  //启动spring容器  ApplicationContext context =  new ClassPathXmlApplicationContext("applicationContext.xml");    //获取对象  Calendar calendar = (Calendar) context.getBean("calendar");  System.out.println(calendar.getTime());  } |

4.7.3.通过实例方法创建对象

步骤：首先创建实例工厂类

|  |
| --- |
| package factory;  import java.util.Calendar;  public class NewInstanceFactory {  public NewInstanceFactory(){  System.out.println("我是实例工厂的构造方法");  }  public Calendar getCalendar(){  return Calendar.getInstance();  }  } |

配置applicationContext.xml

|  |
| --- |
| <!-- 通过实例工厂获取对象 -->  <!-- 创建实例工厂对象 -->  <bean id="newInstanceFactory" class="factory.NewInstanceFactory"></bean>  <!-- 通过实例工厂创建对象  factory-bean:实例工厂的id  factory-method:实例工厂中获取对象的方法名称 -->  <bean id="calendarA" factory-bean="newInstanceFactory"  factory-method="getCalendar"></bean> |

最后进行测试

4.7.4.spring工厂创建对象

Spring提供的创建bean的方法

步骤：首先，创建spring工厂类

|  |
| --- |
| package factory;  import java.util.Calendar;  import org.springframework.beans.factory.FactoryBean;  public class SpringFactory implements FactoryBean<Calendar>{    //最重要的方法，用来获取对象的，必须写  @Override  public Calendar getObject() throws Exception {  return Calendar.getInstance();  }    //获取对象的类型  @Override  public Class<?> getObjectType() {  return Calendar.class;  }    //是否是单例的  @Override  public boolean isSingleton() {  return false;  }  } |

配置applicationContext.xml

|  |
| --- |
| <!-- 配置spring工厂 id:生成对象的id class:spring工厂类的class --> <bean id="calendarB" class="factory.SpringFactory"></bean> |

最后测试

4.8.spring中的单例和多例

单例：在内存中对象有且只有一份

问题：spring中的对象是单例的还是多例的？

测试：

|  |
| --- |
| @Test  public void test07(){  ApplicationContext context =  new ClassPathXmlApplicationContext("applicationContext.xml");  Hello hello = (Hello) context.getBean("hello");  Hello hello1 = (Hello) context.getBean("hello");  System.out.println(hello);  System.out.println(hello1);  System.out.println(hello.getA());  hello.setA(3);  System.out.println(hello1.getA());  } |

测试发现，在spring中维护的对象是单例的，便于维护

如何改变对象的单例多例？

在配置文件中修改：

|  |
| --- |
| <!-- 单例多例修改  scope="singleton":表示单例  scope="prototype":表示多例  -->  <bean id="hello" class="pojo.Hello" scope="prototype"></bean> |

想一想：（在spring容器中，一个key对应一个value，但是要想一个key对应多个value，是不是不好实现）spring如何维护多例（每获取一个key都是一个全新的对象）

说明：在spring容器中，如果是多例的，则对象在容器启动的时候不创建，每次客户端调用的时候，new出一个新的对象，spring容器只管创建，不管维护。（单例和多例分别打断点测试）

4.9.懒加载

如果所有的bean都在容器启动的时候创建，这样性能会下降。

懒加载：什么时候需要，什么时候创建。

例子：首先创建User类，加入默认构造方法

然后配置bean

最后测试，只启动容器

发现默认构造打印出了，说明容器启动，对象就会创建。

|  |
| --- |
| <!-- 测试懒加载  lazy-init="true":懒加载生效  lazy-init="false":懒加载不生效  lazy-init="default":懒加载默认  -->  <bean id="user" class="pojo.User" lazy-init="default"></bean> |

default是与全局变量保持一致，在头文件中配全局懒加载：

|  |
| --- |
| <?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-3.2.xsd"  default-lazy-init="true"> |

注意：如果配置为多例，默认就是懒加载。

4.10.自定义的初始化和销毁方法

当我们在做架构的时候，由于有的类在创建的时候，需要初始化一些操作，比如流的开启，一些链接操作或者是初始化赋值等等，而在容器销毁的时候，要进行一些流的关闭，或者是链接的关闭，资源的释放等操作，这时候就用到了初始化方法和销毁方法。

步骤：首先，在User类中加入初始化和销毁方法

|  |
| --- |
| public void init(){  System.out.println("我是初始化方法");  }  public void destroy(){  System.out.println("我是销毁方法");  } |

在配置文件中配置bean，并加入初始化和销毁方法

|  |
| --- |
| <!-- 测试初始化和销毁 -->  <bean id="user" class="pojo.User"  init-method="init" destroy-method="destroy"></bean> |

最后进行测试

|  |
| --- |
| //测试初始化和销毁  @Test  public void test09(){  //启动spring容器,由于接口中没有关闭容器的方法，所以用实现类  ClassPathXmlApplicationContext context =  new ClassPathXmlApplicationContext("applicationContext.xml");    //获取对象  User user = (User) context.getBean("user");  //只有在容器关闭的时候才会启动销毁方法  context.close();  } |

整个spring中对象的简单生命周期分为4个部分：

1. 对象实例化
2. 对象初始化
3. 对象完成特定的操作
4. 对象销毁

整个spring中，对象的生命周期经历了12个过程，如果有兴趣，自己下去了解

## 1.5.Spring中的DI（依赖注入）

（广泛上讲，DI是ioc的一种，侧重于过程）

依赖注入的方式：1.set方法注入。2.构造方法注入

5.1.set方法注入：（必须要有set方法）

Set方法属性注入

步骤：首先，创建一个person类，添加get与set方法，添加toString

|  |
| --- |
| public class Person {  private Integer id;  private String name;  private List list;  private Set set;  private Map map;  private Properties pro; |

在配置文件中为属性赋值（借助set方法）

|  |
| --- |
| <!-- 配置属性注入 -->  <bean id="person" class="pojo.Person">  <property name="id" value="3"></property>  <property name="name" value="高手"></property>  <property name="list">  <list>  <value>吃饭</value>  <value>睡觉</value>  <value>打豆豆</value>  </list>  </property>  <property name="set">  <set>  <value>被打</value>  <value>被打</value>  <value>被打</value>  <value>被打</value>  </set>  </property>  <property name="map">  <map>  <entry key="一" value="上课"></entry>  <entry key="二" value="逃课"></entry>  <entry key="三" value="被老师批评"></entry>  <entry key="四" value="好好学习"></entry>  </map>  </property>  <property name="pro">  <props>  <prop key="a">打怪</prop>  <prop key="b">回家</prop>  <prop key="c">刷野</prop>  </props>  </property>  </bean> |

测试

|  |
| --- |
| Person person = (Person) context.getBean("person");  System.out.println(person); |

set方法为引用赋值

步骤：首先，创建Dog和Cat两个类，并在Person中引用

|  |
| --- |
| public class Cat {  public void say(){  System.out.println("喵喵");  }  }  public class Dog {  public void say(){  System.out.println("汪汪");  }  }  public class Person {  private Dog dog;  private Cat cat; |

在配置文件中配置Dog和Cat，并在person中引用

|  |
| --- |
| <!-- 为引用类型赋值 ref：引用类型的id -->  <property name="dog" ref="dog"></property>  <property name="cat" ref="cat"></property>  </bean>    <bean id="dog" class="pojo.Dog"></bean>  <bean id="cat" class="pojo.Cat"></bean> |

最后测试

|  |
| --- |
| //测试引用类型注入  person.getDog().say();  person.getCat().say(); |

## 1.6.spring中ioc和DI整合案例

步骤：创建User类

|  |
| --- |
| public class User {  private Integer id;  private String name; |

创建三层结构：UserDao和UserDaoImpl

|  |
| --- |
| public class UserDaoImpl implements UserDao{  private User user;  public void setUser(User user) {  this.user = user;  }  @Override  public void addUser() {  System.out.println("添加一个新用户："+user.getId()+":"+user.getName());  } |

三层结构UserService和UserServiceImpl

|  |
| --- |
| public class UserServiceImpl implements UserService {    private UserDao userDao;  public void setUserDao(UserDao userDao) {  this.userDao = userDao;  }    @Override  public void addUser() {  userDao.addUser();  } |

UserServlet

|  |
| --- |
| public class UserServlet {  private UserService userService;  public void setUserService(UserService userService) {  this.userService = userService;  }    public void addUser(){  userService.addUser();  }  } |

配置文件

|  |
| --- |
| <bean id="user" class="pojo.User">  <property name="id" value="1"></property>  <property name="name" value="乔丹"></property>  </bean>  <bean id="userDao" class="dao.UserDaoImpl">  <property name="user" ref="user"></property>  </bean>  <bean id="userService" class="service.UserServiceImpl">  <property name="userDao" ref="userDao"></property>  </bean>    <bean id="userServlet" class="web.UserServlet">  <property name="userService" ref="userService"></property>  </bean> |

最后测试

|  |
| --- |
| @Test  public void test(){  ApplicationContext context =  new ClassPathXmlApplicationContext("applicationContext.xml");    UserServlet userServlet = (UserServlet) context.getBean("userServlet");  userServlet.addUser();  } |

Spring的ioc和DI可以大大减少程序之间的耦合。

Spring简化了对象的创建