# Spring

## 搭建环境

### 概念

Spring 是一个IoC容器

IoC（Inversion of Control）控制反转

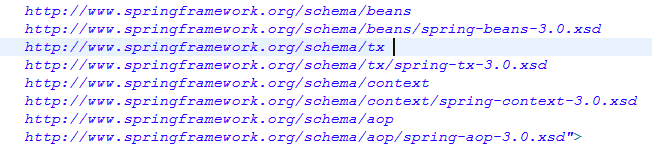
DI （Dependency Injection）依赖注入

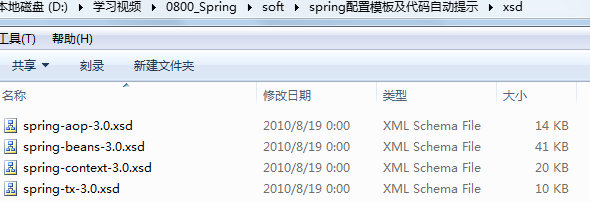
Spring也是一个对象工厂，负责创建和管理各种对象

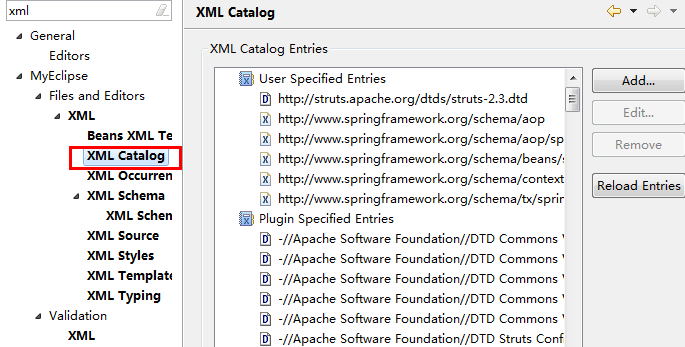
### 拷贝jar包，创建配置文件

### 创建Action Service ServiceImpl

### 自动提示







## 基础知识

### 配置applicationContext.xml

<bean id="userAction" name=”ua”

class="com.spring.actions.UserAction"></bean>

<bean id="userService" name=”us”

class="com.spring.service.impl.UserServiceImpl"></bean>

Spring配置文件不允许出现两个相同id的<bean>，但是可以出现两个相同name的<bean>

Class 是具体类的路径

可以使用一个或多个<import>元素从另外一个或多个文件中加载Bean定义，这样可以把Bean进行分类，定义在不同的配置文件中，所有的<import>必须在<bean>之前完成导入

<beans>

<import resource=”services.xml” />

<bean id=”” class=”” />

</beans>

### 实例化BeanFactory的方式

#### 从文件系统资源文件实例化BeanFactory

//由bean.xml生成Resource实例

Resource res = new FileSystemResource(“bean.xml”);

//生成BeanFactory实例

BeanFactory factory = new XmlBeanFactory(res);

#### 从classpath下的资源实例化BeanFactory

Resource res = new ClassPathResource(“bean.xml”);

BeanFactory factory = new XmlBeanFactory(res);

#### 使用ApplicationContext从classpath下的XMl文件实例化BeanFactory

BeanFactory factory = new

ClassPathXmlApplicationContext("applicationContext.xml");

### 对基本语法的理解

//创建Spring容器

BeanFactory factory = new ClassPathXmlApplicationContext

("applicationContext.xml");

//通过类型获取对象，不需要转型

UserAction userAction = factory.getBean(UserAction.class);

//通过id获取对象，需要转型

UserAction userAction=(UserAction)factory.getBean("userAction");

//通过名称获取对象，需要转型

UserAction userAction=(UserAction)factory.getBean("ua");

userAction.add(); //调用add()方法

//获取接口类型的对象，会自动找到它的实现类

UserService userService =

(UserService)factory.getBean(UserService.class);

userService.add();

也可以通过别名，ID获取接口类型的对象。

### ApplicationContext与BeanFactory的区别联系

ApplicationContext接口由BeanFactory派生而来，用法类似：

ApplicationContex context = new ClassPathXmlApplicationContext

("applicationContext.xml");

UserAction userAction=

(UserAction)context.getBean("userAction");

表面上看ApplicationContex与BeanFactory一样，都是载入bean定义，绑定，最后销毁，但ApplicationContext还提供了一些支持：对In18的支持，一般属性资源文件的读取，给事件提供了监听器支持

最大的区别是：

BeanFactory在调用getbean()时载入；

ApplicationContext在初始化时载入所有的单例Bean

ApplicationContext能利用Java反射机制自动判断出配置文件的定义，并自动将其注册到应用上下文中，而BeanFactory需要手动调用addBeanPostProcessor()方法进行注册

### Scope（都是指在Spring容器中，默认是singleton）

singleton 单例，每次getbean都是保证容器中是同一个对象

prototype 多例，每次getbean都创建一个新的对象

singleton的对象是在创建容器的时候创建

prototype 是在getbean的时候创建

scope=”prototype” 将bean返回给调用者，调用者负责bean后续的管理，Spring不再管理这些bean的生命周期

scope=”singleton” 则将bean放在Spring IoC 容器的缓冲池中，并将bean引用返回给调用者，Spring继续对这些bean进行后续的生命管理

### 初始化Bean

一个Bean可能在实例化之前需要设定某些参数，可以通过以下方式：

在配置文档中指定Init-method

<bean id=”life” class=”” init-method=”init”></bean>

在实体类中有public void init(){ }

实现InitializingBean接口,并增加实现afterPropertiesSet()方法

public class Life implements InitializingBean{

/\*其他方法\*/

public void afterPropertiesSet(){

}

}

当这个Bean的所有属性被Spring的BeanFactory设置完后，会自动调用afterPropertiesSet方法对Bean初始化

### 销毁Bean

在配置文档中指定destroy-method

<bean id=”life” class=”” destroy-method=”destroy”></bean>

实现DisposableBean接口，实现destroy方法

Spring会自动调用Bean的destory方法

### IoC DI 控制反转 依赖注入

控制反转和依赖注入的具体含义：当某个角色需要另一个角色的协助时，在传统程序设计过程中，通常用调用者来创建被调用者的实例。但在Spring中，创建被调用对象的工作不再由调用者完成而是由Spring完成，因此称为控制反转。把创建好的实例注入给调用者，因此也称为依赖注入。

#### 属性注入（设值注入）

需要有对应的setter和默认的构造方法

为什么需要setter？

把Spring自动创建的对象set到调用类中

配置文件：

<bean id="userAction" name="ua" scope="singleton" class="com.spring.actions.UserAction">

//property 配置的是要注入的对象，name对应userAction里的属性名，ref对应被调用类的ID，bean与bean之间的依赖关系也是通过id属性建立

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

</bean>

<bean id="userService" name="us" scope="prototype" class="com.spring.service.impl.UserServiceImpl">

</bean>

由容器创建userService 对象

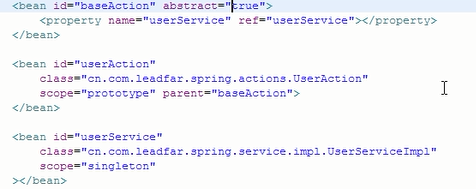
UserService userService ;

//要设置对应的setter方法，把userService注入到UserAction中

public void setUserService(UserService userService) {

this.userService = userService;

}



可以把注入放在BaseAction里，

abstract=”true” 表示这个类不需要创建实例

parent=”baseAction” 表示继承baseAction

<bean id="baseAction" abstract="true">

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

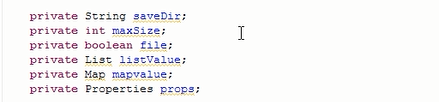
</bean>

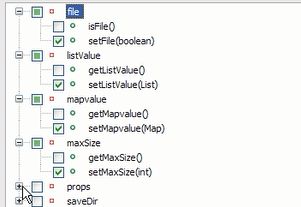
<bean id="userAction" name="ua" scope="singleton" class="com.spring.actions.UserAction" parent="baseAction">

</bean>

#### Bean中普通属性的注入

要有对应的setter方法









#### 用构造方法注入：此时不需要setter方法

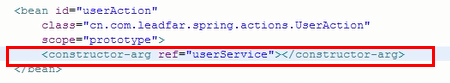
public UserAction (UserService userService){

this.userService = userService;

}

相应的配置为：

不需要property



通过配置文件注入参数，必须与构造方法形参匹配

<constructor-arg><value><value>…..</value> </ constructor-arg >

按类型匹配入参：

<construct-arg type=”java.lang.String”>

<value>……. </value>

<construct-arg type=”int”>

<value>……. </value>

</construct-arg>

按索引匹配入参：

<construct-arg index=”0” value=”……”></construct-arg>

联合类型和索引入参：

<construct-arg type=”” index=”0” >

<value>……..</value>

</construct-arg>

#### 构造方法注入和设值注入的区别

设值注入是在需要某一个bean实例时创建一个默认的bean实例，然后调用对应的设值方法注入依赖关系，而构造方法注入则在创建bean实例时就已经完成了依赖关系的注入

对于依赖关系无须变化的注入尽量采用构造注入，而对于其他的依赖关系的注入，则考虑设值注入

#### Scope与DI

（1）多例注入到单例中有问题

（2）多例注入到多例中没有问题

（3）单例注入到单例中没有问题

（4）单例注入到多例中没有问题

解决第一种情况的办法是，不要用注入，用getBean的方式，同时要保证是同一个容器

public class UserAction implements BeanFactoryAware

private BeanFactory factory;

public void setBeanFactory(BeanFactory factory) throws BeansException {

this.factory = factory;

}

factory.getBean(“userSrvice”);

### 自定义类型转换器

<!-- 注册类型转换器 -->

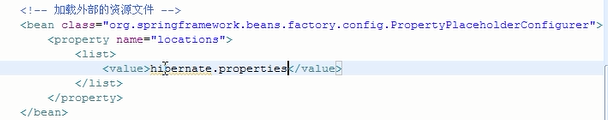
不需要写Id

定义类型转化器

整个过程说明：

读取配置文件，创建userAction对象，创建userService对象，读取配置文件中birthday参数，执行格式化，注入到birthday属性中，把整个userService对象注入到userAction 中，调用add()

### 加载外部资源文件

可以通过${ }引用该配置文件里的参数

<bean id=”” class=””>

<property name=”name” value=”${ drivername}”

</bean>

hibernate.properties配置信息：

drivername=” drivername123”

### 自动注入

需要setter



按照名称自动注入，自动调用setter，属性名必须跟bean的Id一致，此时可以去掉<property name="userService" ref="userService"></property>

改为：<bean id=”” class=”” autowire=”byName”>

按照类型自动注入autowire=”byType” ，此时多个相同类型的对象则不行



autowire=”deFault”，默认值，与根元素定义的default-autowire一致

### 利用注解注入

不需要setter，最常用的，不能注入普通属性，要注入普通属性还需要写配置文件

需要JDK 1.6 及以上版本

定义context：component-scan，指定spring要扫描哪些包

在类的前面写@Component来定义bean

在需要注入的属性前面用@Resource来进行注入

定义<context:component-scan base-package="com.spring">

</context:component-scan>指定Spring要扫描哪些包

不需要配置文件定义<bean></bean>

需要容器创建哪个类就在类前面加上@Component(“xXX”)



括号中是给该类指定的ID，默认定义的Bean的ID是类名（第一个字母小写），此时就不需要setter了

@Component

public class UserAction{…………}

需要注入的属性前加@Resource，指明把该对象注入到本类中，默认是按照类型进行注入，可以指定按照名称进行注入

@Resource 默认按类型

定义按名称注入



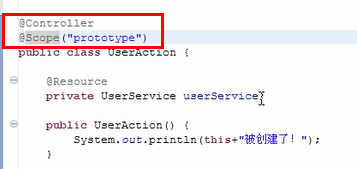
加载到容器中也可以用如下定义：

@Controller定义呈现层 Action

@Service定义业务逻辑对象 Service

@Repository定义持久化层 Dao

@Scope(“prototype”) 定义Scope属性，默认是singleton



注释和配置文件是通用的

### 分离关注点

用静态代理

好处：可以专心做一件事，其余由静态代理完成

坏处：会产生大量重复代码

动态代理

### AOP



切面（aspect）：是指需要实现的交叉功能。

连接点（joinpoint）：应用程序执行过程中插入切面的地点

通知（advice）：是切面的实际实现

切入点（pointcut）：定义了通知应该应用在哪些连接点

目标对象（target object）：是被通知对象

AOP代理（AOP Proxy）：是将通知应用到目标对象后创建的对象

分类：前置通知，后置通知，异常通知，环绕通知，引入通知

BeforeAdvice 前置通知

BeforeAdvice是前置通知接口，而MethodBeforeAdvice接口是其子类

public class ConcreteReception implements Reception{

public void serveCustomer(String customerName){

sysout

}

}

public class GreetingBeforeAdvice implements MethodBeforeAdvice{

public void before (Method method,Object[] args,

Object obj)throws Throwable{

String customerName=(String)args[0];

sysout

}

}

AfterAdvice后置通知

public class GreetingAfterAdvice implements AfterReturningAdvice {

public void afterReturning (Object returnObj,Method method,Object[] args,Object target)throws Throwable{

String customerName=(String)args[0];

sysout

}

}

配置：

<bean id=” greetingBeforeAdvice” class=” GreetingBeforeAdvice”>

<bean id=” greetingAfterAdvice” class=” GreetingAfterAdvice”>

<!—使用Spring代理工厂配置一个代理-->

<bean id=”reception” class=”org.springframework.aop.framework.ProxyFactoryBean”>

<!—指定代理接口，如果是多个代理可以使用<list>元素指定 -->

<property name=”proxyInterfaces” value=”Reception”/>

<!—指定通知-->

<property name=”interceptorNames”>

<list>

<value> greetingBeforeAdvice </value>

<value> greetingAfterAdvice </value>

</list>

</property>

<property name=”target” ref=”target”/>

</bean>

<bean id=”target” class=” ConcreteReception”/>

调用：

ApplicationContext ctx = new ClassPathXmlApplicationContext(beans.xml);

Reception reception = (Reception) ctx.getBean(“reception”);

reception.saveCustomer(“acc”);

ProxyFactoryBean是FactoryBean接口的实现类，负责为其他的Bean创建代理实例，一些属性的解释

target代理的目标对象

proxyInterfaces代理实现的接口

interceptorNames需要织入目标对象的通知类数组，配置中的顺序决定调用的顺序

singleton返回的代理是否为单例，默认为单例

optimize设置为true时，强制使用CGLIB代理，对于singleton推荐使用CGLIB代理，对于其他作用域代理最好使用JDK代理，因为CGLIB代理创建速度慢但是运行效率高，JDK代理相反

proxyTargetClass是否对类进行代理，而不是对接口进行代理，设置为true时，使用CGLIB代理

AroundAdvice环绕通知

public class GreetingAroundAdvice implements MethodInterceptor{

public Object invoke(MethodInvocation invocation)throws Throwable{

//获得目标方法的入参数组

Object args[] = invocation.getArguments();

sysout(“欢迎”);

通过proceed方法反射调用目标实例相应的方法

Object obj = invocation.proceed();

sysout(“请慢走”);

return obj;

}

}

使用MethodInteceptor可以返回一个与proceed()返回的对象完全不同的对象，使用AfterReturningAdvice不能返回一个不同的对象

配置：

<bean id=” greetingAroundAdvice” class=” greetingAroundAdvice”>

<!—使用Spring代理工厂配置一个代理-->

<bean id=”reception” class=”org.springframework.aop.framework.ProxyFactoryBean”>

<!—指定代理接口，如果是多个代理可以使用<list>元素指定 -->

<property name=”proxyInterfaces” value=”Reception”/>

<!—指定通知-->

<property name=”interceptorNames”>

<list>

<idref local=” greetingAroundAdvice”>

</list>

</property>

<property name=”target” ref=”target”/>

</bean>

<bean id=”target” class=” ConcreteReception”/>

加入三个jar包

定义一个拦截器：

@Component

public class TransactionManager implements MethodInterceptor {

public Object invoke(MethodInvocation invocation) throws

Throwable {

System.out.println("拦截之前");

Object value = invocation.proceed();

System.out.println("拦截之后");

return value;

}}

配置：

<aop:config> advice-ref表示用什么拦截，pointcut表示拦截什么

<aop:advisor advice-ref="transactionManager" pointcut=

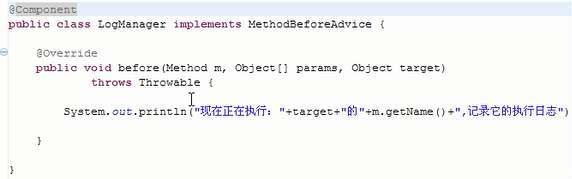
"execution(\* com.spring.service.\*.\*(..))" />

</aop:config>

第一个\* 表示有没有返回值类型，第二个\* 表示任何类，第三个\* 表示任何方法，括号里的点表示无论有没有参数

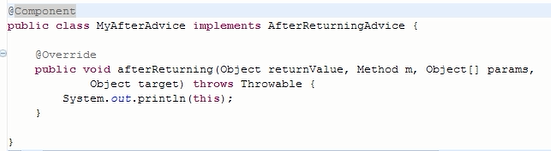
在事务中加入日志管理：

执行方法之前调用到BeforeManager，重写before方法，不必指定接下来让它执行什么，会默认向下执行



m表示调用的方法，params 表示方法携带的参数，target表示所属的对象

定义在方法调用之后执行什么重写afterReturning方法

配置：



可以通过order控制执行顺序

默认编号是transactionManager ，beforeManager ，afterManager 分别为1，2，3，方法之前是从小到大执行，方法执行之后是从大到小执行transactionManager beforeManager afterManager transactionManager

execution(\* com.spring.service.\*.\*(..))会为指定目录下的对象创建动态代理

### SSH集成

#### 创建一个web project

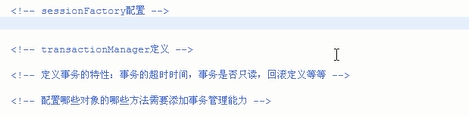
#### 拷贝jar包

除了Struts hibernate Spring 的依赖包之外需要拷贝一个Struts-Spring集成的依赖包，去掉重复的

把SessionFactory，session，transaction都交给Spring管理

要点：理解sessionFactory、transactionManager、txAdvice、AOP配置之间的联系

#### 配置ApplicationContext.xml





用transactionManager管理SessionFactory，相当于让transactionManager管理多个Session，而又可以理解为一个Session对应一个transaction，所以transactionManager管理多个transaction

#### 创建一个实体类User，定义User.hbm.xml 改

hibernate.cfg.xml，创建数据库ssh

定义接口UserDao，定义实现类UserDaoImpl，并注册到Spring中，注入sessionFactory

sessionFactory.getCurrentSession()取得当前的session对象，因为SessionFactory已经交给Spring管理，所以不能openSession，不用提交和回滚事务，同时也不需要手动关闭session

定义UserService接口，定义实现类UserServiceImpl，并注册到Spring中，注入userDao

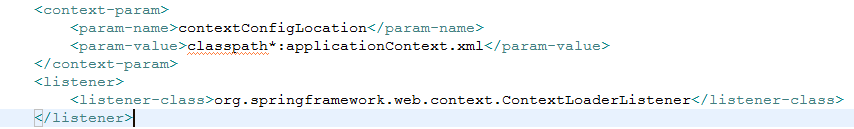
不能在Dao层插入数据，因为配置的拦截是UserService，如果拦截Dao层事务范围太小，一个Service会调用到多个Dao，这样可以保持Dao都在一个事务中

#### 集成Struts2和Spring

配置web.xml ，struts.xml，配置listener

Opentype怎么打开的 Ctrl+Shift+T

配置Listener，使得应用服务器在启动的时候创建Spring容器，并且把BeanFactory对象放在ServletContext中



#### 写Action，并注册到容器中，设置成多例

#### 配置Struts.xml，写jsp 页面

UnsupportedClassVersionError：bad version number in .class file 这是JDK版本的问题，开发环境的JDK版本要与运行环境的JDK版本一致

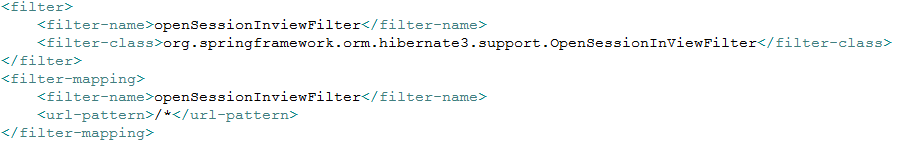
#### 增加查询页面，

User dbuser = userSrvice.findUser(user.getId())

ActionContext.getContext().put(“user”,dbuser)

#### 懒加载异常

在配置AOP中，方法调用完成后transaction就提交，Session也关闭，而使用load()时Session必须开启，要是session延迟到jsp呈现完成之后在关闭，解决办法，在web.xml中加一个openSessionInviewFilter，要放在Struts2 filter前面



#### 测试单元出现懒加载异常

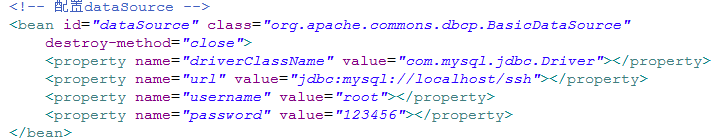
使用load()时Session必须开启，要是session延迟到测试方法完成之后在关闭，解决办法是在测试单元继承一个testCase的实现类，就不需要自己打开容器和getbean了

#### 把hibernate配置文件配置信息转移到Spring配置文件中

定义数据源(dbcp) ，hibernate数据库连接部分配置成dataSource ，再把datasource和其他配置配到SessionFactory中，加入dbcp数据源的依赖包 commons-dbcp和commons-pool

更改applicationContext.xml

destroy-method=”close”是FactoryBean关闭时释放数据库连接资源







某个方法定义成不回滚



测试单元默认会事务回滚，即测试时对数据库不做更改

defaultRollback=false 定义成不回滚

#### 定义事务特性

默认情况下，抛出RuntimeException及其子类型将导致事务回滚，其他异常不会回滚，用rollback-for控制

一般把查询方法定义成read-only=true，增删改保持默认的

read-only=false

#### 错误处理

报错信息：No bean named 'sessionFactory' is defined

原因：在web.xml中加载applicationContext.xml文件的时候没有加载成功，看路径是否正确，这个配置错误，服务器启动的时候不会报错