# 1. Struts2整合的分析

## 1.1 Struts中对象的创建

其实Struts框架不仅仅是一个MVC框架，该框架本身也有如下特点：

Struts2具有插件机制，是一个可插拔的框架；

Struts2也应用了依赖注入和AOP。

下面进行具体解释（为整合Spring做准备）。

之前我们说（或者通过查看源码），Struts启动时（即StrutsPrepareAndExecuteFilter过滤器的init方法中），会按序加载如下文件：

（1）default.properties文件，在org/apache/struts2路径下，其中描述了Struts2中的一些常量信息；

（2）再按序加载以下文件：

struts-default.xml：核心的配置文件；

struts-plugin.xml：插件的配置文件；

struts.xml：开发者编写的配置文件。

说明：上述三个文件必须在classpath根目录下才能加载，且这三个配置文件的dtd约束都是一样的，说明结构相同，如果出现相同的元素，则后者设置会覆盖前者设置。

struts-plugin.xml文件一般都会出现在struts的插件包中，struts框架会加载到（因为jar包在classpath目录，其中的struts-plugin.xml也在classpath下）。在struts开发包的lib目录下，有很多插件包（文件名中包含plugin的包），其中几乎每个插件包都有包含了struts-plugin.xml文件，就用于配置Struts。

在Struts框架中有个叫ObjectFactory（对象工厂）的类，其中有一些方法专门用于创建Struts需要使用的对象，例如buildAction方法、buildInterceptor方法和buildResult等方法，它们分别用于创建Action对象、拦截器对象和结果集对象等。也就是说Struts启动时要利用ObjectFactory创建对象。

Struts2还有静态注入的概念，静态注入即只在初始时向Struts容器中注入bean。例如在struts-default.xml文件中，bean标签中类都将被静态输入：

|  |
| --- |
| <**bean class="com.opensymphony.xwork2.ObjectFactory" name="struts"**/> <**bean type="com.opensymphony.xwork2.factory.ResultFactory" name="struts" class="org.apache.struts2.factory.StrutsResultFactory"** />  *<!-- 其他bean...... -->* |

例如上面中第一个bean就指定把ObjectFactory加入了bean。当Struts启动时就会加载bean。因此最终Action等产生的方式就是通过这样创建出来的（Struts容器创建好bean中的ObjectFactory，然后利用ObjectFactory创建一些需要的对象如Action）。

为了验证上述的说法正确，我们自行“改变”Action的创建方式。

（1）首先编写一个MyObjectFactory，继承ObjectFactory，并重写buildAction方法，这里我们直接返回null：

|  |
| --- |
| **package** com.ssh.factory;  **import** com.opensymphony.xwork2.ObjectFactory; **import** com.opensymphony.xwork2.config.entities.ActionConfig;  **import** java.util.Map;  **public class** MyObjectFactory **extends** ObjectFactory {  @Override  **public** Object buildAction(String actionName, String namespace, ActionConfig config, Map<String, Object> extraContext) **throws** Exception {  **return null**;  } } |

（2）在自行编写的struts.xml文件中覆盖配置ObjectFactory和struts.objectFactory常量：

|  |
| --- |
| <**bean type="com.opensymphony.xwork2.ObjectFactory" name="myObjectFactory" class="com.ssh.factory.MyObjectFactory"** /> *<!-- 下面的value和上述的name一致 -->* <**constant name="struts.objectFactory" value="myObjectFactory"** /> |

说明：struts.objectFactory常量的配置就是设置Struts引用的对象工厂ObjectFactory的bean。

这样配置后，启动Tomcat访问Action时，就会报空指针异常（NPE），因为创建Action时总是返回null。

## 1.2 Struts与Spring的整合

Struts与Spring的整合就是把Action对象的创建交给Spring。在整合Spring与Struts时，需要在Web项目中引入Struts类库中提供的一个插件包“struts2-spring-plugin-2.5.14.1.jar”，该包的根目录下就有一个“struts-plugin.xml”文件，有这样的配置：

|  |
| --- |
| <**bean type="com.opensymphony.xwork2.ObjectFactory" name="spring" class="org.apache.struts2.spring.StrutsSpringObjectFactory"** /> <**constant name="struts.objectFactory" value="spring"** /> |

上述配置的作用就是：重新定义了ObjectFactory，让struts2的对象工厂由Spring创建。这样Struts中的Action等对象就会由Spring创建。当Struts启动后，Struts就会读取struts-plugin.xml中的内容并应用。

所以这就是插件的机制：只要在项目中引入插件包，就能对Struts进行一些变更或扩展，如果不想要插件提供的功能，只要在项目中去掉插件包即可。

实际上，把框架底层的原理弄清楚后，使用框架基本就会得心应手。现在开发者更倾向使用SpringMVC而不是Struts，也可能是对Struts2了解不是很深。如果想要开发时灵活可控，甚至可以采用最纯粹的Servlet规范中的东西，可随意按照需求灵活使用。因为有些框架做的不灵活的话，想要实现一些东西是不好做的。例如Activiti框架就不是很灵活，因此有些公司使用的就是自己开发的框架，只要能完成自己的业务即可。

# 2. SSH整合

## 2.1 目标分析

SSH整合就是在项目中使用Struts和Hibernate框架完成MVC和数据库的操作，并利用Spring框架进行整合。整合的目的是将项目中主要对象交由Spring来管理，增强项目的扩展性。

SSH整合的关键之处：

（1）数据源dataSource、sessionFactory要由Spring管理，由于要利用Spring做声明式事务处理，则DAO和Service层的类也要交由Spring处理。这里就会完成Spring与 Hibernate的整合。

（2）Struts2中的Action类也要交由Spring管理，因为Action层需要使用Service对象，为了实现完全地面向接口编程和便于依赖注入，Action需要交由Spring管理。这里就完成了Spring与Struts的整合。

下面就来搭建一个稍微完整的SSH框架（会使用注解）。

## 2.2 新建项目

对于集成开发环境IDE，我们推荐使用Eclipse或者Intellij IDEA，而不推荐使用MyEclipse。MyEclipse臃肿庞大，且新建项目时会默认给项目添加内置的应用服务器和JSTL库等，而这些本是应由开发人员自行配置的，这样便于控制项目。

首先新建一个Java Web项目，引入下面包（在WEB-INF/lib目录下）：

（1）JDBC驱动包、Hibernate相关包、druid包（用于数据库连接池）。

（2）Struts相关包、“struts2-spring-plugin-2.5.14.1.jar”包（用于Spring与Struts整合）。

（3）Spring相关包、AOP相关包、commons-logging包、JDBC支持包、tx支持包（事务）、orm包、web支持包（因为这是WEB项目）。

说明，在下面，我们将新建源码包（Source Folder），将配置文件、测试文件都放入其中，结构清晰。例如在Eclipse中，在根目录下再新建Source Folder，名叫test、config等，而在IDEA下，直接新建目录，然后把包标记为“Source”。详见3.4节的单元测试。后续把这两个测试整合到一起。

## 2.3 Service、DAO及其数据库相关整合（Hibernate）

整合项目以操作Student数据库表为例，自动建表。

（1）新建com.ssh.domain包，Student实体类如下：

|  |
| --- |
| **package** com.ssh.domain;  **import** java.io.Serializable; **import** java.util.Date;  **public class** Student **implements** Serializable {  **private int id**;  **private** String **name**;  **private** Date **entranceTime**;  **public** Student() {}  *// 其他构造器、setter/getter/toString等* **public** Student(**int** id, String name, Date entranceTime) {  **this**.**id** = id;  **this**.**name** = name;  **this**.**entranceTime** = entranceTime;  } } |

（2）同包下的student.hbm.xml文件如下：

|  |
| --- |
| *<?***xml version="1.0" encoding="utf-8"** *?>* **<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>*** <**hibernate-mapping package="com.ssh.domain"**>  <**class name="Student" table="student"**>  <**id name="id" column="id"**>  <**generator class="assigned"** />  </**id**>  <**property name="name" column="name"** />  <**property name="entranceTime" column="entranceTime"** />  </**class**> </**hibernate-mapping**> |

（3）初步配置applicationContext.xml。

在src下创建db.properties：

|  |
| --- |
| **driverClassName**=**com.mysql.jdbc.Driver username**=**root password**=**123qwe!@# url**=**jdbc:mysql://114.55.86.230:3306/mytest connectionProperties**=**useUnicode=false;characterEncoding=UTF8;useSSL=false** |

并创建applicationContext.xml文件，内容如下：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:context="http://www.springframework.org/schema/context" xmlns:tx="http://www.springframework.org/schema/tx"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd"**>  *<!-- 引入db.properties配置文件 -->* <**bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"**>  <**property name="locations" value="classpath:db.properties"**/>  </**bean**>  *<!-- 配置DataSource(使用druid) -->* <**bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource" init-method="init" destroy-method="close"**>  *<!-- 连接池的属性 ...... -->*  </**bean**>   *<!-- 配置sessionFactory -->* <**bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean"**>  *<!-- 其中引用dataSource -->* <**property name="dataSource" ref="dataSource"** />  *<!-- 映射文件所在的路径 -->* <**property name="mappingDirectoryLocations"**>  <**list**>  <**value**>classpath:com/ssh/domain</**value**>  </**list**>  </**property**>  *<!-- hibernate其他属性 -->* <**property name="hibernateProperties"**>  <**props**>  <**prop key="hibernate.dialect"**>org.hibernate.dialect.MySQL5Dialect</**prop**>  <**prop key="hibernate.show\_sql"**>true</**prop**>  <**prop key="hibernate.format\_sql"**>true</**prop**>  <**prop key="hibernate.hbm2ddl.auto"**>update</**prop**>  *<!-- 线程方式得到session(可省略) -->* <**prop key="hibernate.current\_session\_context\_class"**>org.springframework.orm.hibernate5.SpringSessionContext</**prop**>  </**props**>  </**property**>  </**bean**>   *<!-- 开启注解扫描 -->* <**context:component-scan base-package="com.ssh"** />  *<!-- 以下是用注解进行声明式事务管理所需的XML配置 -->   <!-- 配置hibernate事务管理器 -->* <**bean id="transactionManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager"**>  *<!-- 引用sessionfactory -->* <**property name="sessionFactory" ref="sessionFactory"** />  </**bean**>  *<!-- 开启自动创建声明式事务 -->* <**tx:annotation-driven transaction-manager="transactionManager" proxy-target-class="true"** /> </**beans**> |

（4）开始写DAO层。但这里不直接写StudentDAO，而是先写一个通用的DAO类。因为每个实体类对应DAO类都有一些通用的增删改查操作，为了避免在每个DAO类中都重复写这些增删改查操作，我们把基础DAO操作抽取出来，形成IBaseDao接口及其实现BaseDaoImpl。为了适应不用的实体类型，BaseDao使用了泛型。

IBaseDao代码（通用DAO接口，在com.ssh.dao中）：

|  |
| --- |
| **package** com.ssh.dao;  **import** java.io.Serializable; **import** java.util.List;  */\*\*  \* 基础DAO接口提供基本操作方法的原型  \*/* **public interface** IBaseDao<T> {  */\*\*  \* 添加一个对象  \** ***@param t*** *对象  \*/* **void** add(T t);   */\*\*  \* 根据主键删除对象数据  \*/* **void** delete(T t);   */\*\*  \* 更新对象数据  \** ***@param t*** *对象  \*/* **void** update(T t);   */\*\*  \* 根据主键查询对象数据  \** ***@param clazz*** *类字节码文件  \** ***@param id*** *主键  \** ***@return*** *对象  \*/* T findById(Class<T> clazz, Serializable id);   */\*\*  \* 查询某实体所有的数据  \** ***@param clazz*** *类字节码文件  \** ***@return*** *对象集合  \*/* List<T> findAll(Class<T> clazz); } |

BaseDaoImpl代码（通用DAO实现，在com.ssh.dao.impl包中）：

|  |
| --- |
| **package** com.ssh.dao.impl;  **import** com.ssh.dao.IBaseDao; **import** org.hibernate.Session; **import** org.hibernate.SessionFactory;  **import** javax.annotation.Resource; **import** java.io.Serializable; **import** java.util.List;  **public abstract class** BaseDaoImpl<T> **implements** IBaseDao<T> {   @Resource(name = "sessionFactory")  **protected** SessionFactory **sessionFactory**;   @Override  **public void** add(T t) {  **sessionFactory**.getCurrentSession().save(t);  }   @Override  **public void** delete(T t) {  **sessionFactory**.getCurrentSession().delete(t);  }   @Override  **public void** update(T t) {  **sessionFactory**.getCurrentSession().update(t);  }   @Override  **public** T findById(Class<T> clazz, Serializable id) {  **return sessionFactory**.getCurrentSession().get(clazz, id);  }   @Override  **public** List<T> findAll(Class<T> clazz) {  Session session = **sessionFactory**.getCurrentSession();  **return** session.createQuery(**"from "** + clazz.getName(), clazz).list();  } } |

说明：

1. 本类使用了abstract关键字修饰了，强制这个类不能被直接使用，必须继承使用。因为直接使用是没有意义的。

2. 本类不需要加入到Spring容器，因为继承本类的DAO类加入Spring容器即可。但其中的sessionFactory属性需依赖注入，否则继承本类的DAO类是不好用注解注入该属性的。并且为了继承本类的DAO类方便使用sessionFactory，把sessionFactory用protected关键字进行修饰。

这样，其他DAO实现类只要继承BaseDaoImpl即可拥有通用的操作的方法。

（5）编写IStudentDao和StudentDaoImpl。

这时，每个DAO接口应该继承通用DAO接口，这样才会有基础方法。接口中只要声明自己特有的方法。因此IStudentDao代码：

|  |
| --- |
| **package** com.ssh.dao;  **import** com.ssh.domain.Student; **import** java.util.List;  *// 这里继承的泛型参数就用Student* **public interface** IStudentDao **extends** IBaseDao<Student> {  */\*\*  \* 按姓名查找学生  \** ***@param name*** *姓名  \** ***@return*** *学生列表  \*/* List<Student> findByName(String name); } |

在DAO实现类中，不仅要继承通用的DAO类，还要实现对应的接口。最后，自己的实现类还要加入到Spring容器，因此StudentDaoImpl代码为：

|  |
| --- |
| **package** com.ssh.dao.impl;  **import** com.ssh.dao.IStudentDao; **import** com.ssh.domain.Student; **import** org.hibernate.query.Query; **import** org.springframework.stereotype.Repository; **import** java.util.List;  @Repository(**"studentDao"**) **public class** StudentDaoImpl **extends** BaseDaoImpl<Student> **implements** IStudentDao {  @Override  **public** List<Student> findByName(String name) {  *// 下面可直接使用sessionFactory* Query<Student> query = **sessionFactory**.getCurrentSession().createQuery(**"from Student where name = :name"**, Student.**class**);  query.setParameter(**"name"**, name);  **return** query.list();  } } |

（6）编写Service层。

先写IStudentService接口，例如现在其中只提供了add、delete和findByName方法：

|  |
| --- |
| **package** com.ssh.service;  **import** com.ssh.domain.Student;  **import** java.io.Serializable; **import** java.util.List;  **public interface** IStudentService {   *// 添加学生* **void** add(Student student);   *// 按主键查询学生* Student findById(Serializable id);   *// 删除学生(根据id)* **void** delete(**int** id);   *// 查询指定姓名学生* List<Student> findByName(String name); } |

实现类中应该加入事务处理：

|  |
| --- |
| **package** com.ssh.service.impl;  **import** com.ssh.dao.IStudentDao; **import** com.ssh.domain.Student; **import** com.ssh.service.IStudentService; **import** org.springframework.stereotype.Service; **import** org.springframework.transaction.annotation.Transactional; **import** javax.annotation.Resource; **import** java.io.Serializable; **import** java.util.List;  @Service(**"studentService"**) *// 加入容器* **public class** StudentServiceImpl **implements** IStudentService {   *// 注入studentDao* @Resource(name = **"studentDao"**)  **private** IStudentDao **studentDao**;   @Override  @Transactional(readOnly = **false**)  **public void** add(Student student) {  **studentDao**.add(student);  }   @Override  @Transactional(readOnly = **true**)  **public** Student findById(Serializable id) {  **return studentDao**.findById(Student.**class**, id);  }   @Override  @Transactional(readOnly = **false**)  **public void** delete(**int** id) {  Student student = **new** Student();  student.setId(id);  **studentDao**.delete(student);  }   @Override  @Transactional(readOnly = **true**)  **public** List<Student> findByName(String name) {  **return studentDao**.findByName(name);  } } |

这样，Service层及其以下的配置均搞好了。

## 2.4 单元测试验证

上述步骤做好以后应该做一个单元测试，以验证上述编写的正确性，确保本部分代码无误。否则一股脑全部整合好后发现问题一大堆，将难以查找错误。因此单元测试也非常重要。

虽然这是在WEB项目中，但Service、DAO等层是不涉及WEB的，因此这里还是可以使用原来的方法进行单元测试：

|  |
| --- |
| **package** com.ssh.test;  **import** com.ssh.domain.Student; **import** com.ssh.service.IStudentService; **import** org.junit.Test; **import** org.springframework.context.support.ClassPathXmlApplicationContext; **import** java.util.Date;  *// Service的单元测试* **public class** TestService {  @Test  **public void** test() {  ClassPathXmlApplicationContext ac = **new** ClassPathXmlApplicationContext(**"applicationContext.xml"**);  IStudentService studentService = (IStudentService) ac.getBean(**"studentService"**);  studentService.add(**new** Student(1, **"张三"**, **new** Date()));  ac.close();  } } |

上述代码运行无错误，且能插入数据到数据库中，说明Spring配置基本正确。为什么运行时能找到我们引入的jar包，那是因为IDE自动把WEB-INF/lib目录加入到了classpath中，这和我们在Java项目中新建一个lib目录并设置为jar包目录是一个道理。

如果applicationContext.xml文件不在src目录下，例如在src/config目录下，则使用：

|  |
| --- |
| ClassPathXmlApplicationContext ac = **new** ClassPathXmlApplicationContext(**"config/applicationContext.xml"**); |

即可。

## 2.5 Spring与Struts的整合

首先编写Action类，这部分和之前写法没有什么不同。只是注意也要将Action类也加入Spring容器中，并且使用多例模式。例如：

|  |
| --- |
| **package** com.ssh.action;  **import** com.opensymphony.xwork2.ActionSupport; **import** com.ssh.service.IStudentService; **import** org.apache.struts2.ServletActionContext; **import** org.springframework.context.annotation.Scope; **import** org.springframework.stereotype.Controller; **import** javax.annotation.Resource;  @Controller *// 将Action加入容器中* @Scope(**"prototype"**) *// 多例模式* **public class** StudentAction **extends** ActionSupport {   @Resource(name = **"studentService"**) *// 注入service* **private** IStudentService **studentService**;   *// 显示一个学生数据* **public** String showStudent() {  *// 把结果保存到request域中* ServletActionContext.*getRequest*().setAttribute(**"student"**, **studentService**.findByName(**"张三"**).get(0));  **return *SUCCESS***;  } } |

由于我们已经引入了struts2-spring-plugin-2.5.14.1.jar这个整合包，因此Action对象的创建会由Spring创建，在编写Struts.xml时，action标签中的class属性值可直接写为Spring中bean的id标识名称，Struts插件包中会自动找到匹配的Action类。即src下的struts.xml配置文件为：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* **<!DOCTYPE struts PUBLIC  "-//Apache Software Foundation//DTD Struts Configuration 2.5//EN"  "http://struts.apache.org/dtds/struts-2.5.dtd"*>*** <**struts**>  <**package name="default" namespace="/" extends="struts-default"**>  <**action name="showStudent" class="studentAction" method="showStudent"**>  <**result name="success"**>/index.jsp</**result**>  </**action**>  </**package**> </**struts**> |

当然，显示的index.jsp页面代码如下：

|  |
| --- |
| <%@**page pageEncoding**="**UTF-8**" **language**="**java**" %> <!doctype **html**> <**html lang="zh-CN"**> <**head**>  <**meta charset="UTF-8"**>  <**title**>学生展示</**title**> </**head**> <**body**>  姓名；**${**student.name**}** 入学时间：**${**student.entranceTime**}** </**body**> </**html**> |

关于Struts中日志文件的配置也可加上，这里就不赘述了，方法和之前一样，复制之前的log4j.xml文件到src下即可。

最后的配置其实就是在web.xml中了，因为要通过Web容器加载Struts过滤器，并且Spring容器也要在Web容器中启动，因此，web.xml中应该这样配置：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"  version="3.1"**>  *<!-- Struts过滤器配置 -->* <**filter**>  <**filter-name**>struts2</**filter-name**>  <**filter-class**>org.apache.struts2.dispatcher.filter.StrutsPrepareAndExecuteFilter</**filter-class**>  </**filter**>  <**filter-mapping**>  <**filter-name**>struts2</**filter-name**>  <**url-pattern**>\*.action</**url-pattern**>  </**filter-mapping**>  *<!-- Spring配置。指明配置文件的位置。 -->* <**context-param**>  <**param-name**>contextConfigLocation</**param-name**>  <**param-value**>classpath:applicationContext.xml</**param-value**>  </**context-param**>  *<!-- Spring的监听器。Web应用中需要配置监听。 -->* <**listener**>  <**listener-class**>org.springframework.web.context.ContextLoaderListener</**listener-class**>  </**listener**> </**web-app**> |

这里的Struts的Action访问路径设置为“\*.action”，表示只有请求路径后缀是“action”的请求才会进入Struts中处理，没必要使用以前的“/\*”，因为那样会把HTML等文件也经过Struts过滤器处理再返回该资源。另外，也可自行配置Struts的常量。

这样，Spring与Struts以及Hibernate的整合就完成了，启动服务器部署应用可以正常访问“showStudent.action”这个URL。

其实对于Struts的整合，只要引入插件包、在struts.xml中使用到class属性时，直接引用Spring中的bean名称即可，最后配置一下web.xml就完成了。

## 2.6 总结

从上述过程看出，搭建环境首先要熟悉其原理，一步一步完成，会发现逻辑清晰，环环相扣，并且这样开发者做事具有效率。

实际上也可以将Spring配置分离成多个文。比如在src下存在bean1.xml和bean2.xml文件，那么定位Spring配置文件时，可使用“classpath:bean\*.xml”。其中的classpath就表示在classpath路径下找到文件。运行时，它们将会是一个整体的配置。

分离Spring配置文件的方式还有“包含”的方式，即类似于Struts的“include”，Spring是通过“import”节点配置导入其他Spring配置文件。其中也能使用通配符，但是通配符不能通配多个文件夹，只能是一个。比如：

|  |
| --- |
| *<!-- 引入外部sprign配置文件 -->* <**import resource="classpath:com/zhang/\*/\*-spring.xml"**/> |

此外，Java Web项目可以打包成war包，可以直接将war包复制到Tomcat/webapps目录下，当启动Tomcat服务器时，服务器会自动解压war包并加载此项目。用war包进行部署非常方便。

使用Spring容器注入对象时，一定要理清各个对象之间的依赖关系，这样才能正确地进行管理。否则一旦配置错误，某个对象无法拿到依赖的属性，则会有空指针异常、不能注入等问题。

# 3. SSH整合的原理

## 3.1 Spring容器的创建

在web.xml中，为了整合Spring，我们注册了ContextLoaderListener监听器，该监听器主要就是用来创建Spring容器的。

（1）首先，这个类是Spring的web模块提供的，ContextLoaderListener是实现了Servlet规范中的ServletContextListener接口的监听器。其中的contextInitalized和contextDestroyed方法分别是创建Spring容器和销毁Spring容器。也就是说，WEB中Spring容器的生命周期就是在servletContext范围中。

（2）ContextLoaderListener的contextInitalized方法调用了ContextLoader类的initWebApplicationContext方法。

1. 首先我们分析ContextLoader类。

在该类的静态代码块中（即142行左右），使用“DEFAULT\_STRATEGIES\_PATH”常量创建了一个resource资源对象，并用其得到了defaultStrategies对象。这个DEFAULT\_STRATEGIES\_PATH常量的值在该类中就是“ContextLoader.properties”，同样在该类路径下也发现了该配置文件，而ContextLoader.properties只有一行配置：

|  |
| --- |
| **org.springframework.web.context.WebApplicationContext**=**org.springframework.web.context.support.XmlWebApplicationContext** |

也就是说加载了这个文件作为创建Spring容器的策略，properties文件指定使用XmlWebApplicationContext这个类来创建ApplicationContext（即Spring容器）。而我们之前的非WEB是采用ClassPathXMLApplicationContext。XmlWebApplicationContext就是ApplicationContext的一个实现类。

也就是说，static静态代码中就是根据配置确定使用哪个Spring Web容器。只要调用这个类的方法就会执行static代码块。

2. 再分析调用的initWebApplicationContext方法。

initWebApplicationContext方法就是用来初始化WEB容器的。在本方法开头（261行）就有这样的代码：

|  |
| --- |
| **if** (servletContext.getAttribute(WebApplicationContext.***ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE***) != **null**) {  **throw new** IllegalStateException(  **"Cannot initialize context because there is already a root application context present - "** +  **"check whether you have multiple ContextLoader\* definitions in your web.xml!"**); } |

其中的“WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE”常量值被赋值为：

|  |
| --- |
| String ***ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE*** = WebApplicationContext.**class**.getName() + **".ROOT"**; |

也就是说WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE的值就是“WebApplicationContext.ROOT”。

那么这个261行开始的代码的意思就是从servletContext中拿到键为“WebApplicationContext.ROOT”的对象，如果该值不为null，则出现异常，否则正常进行，下面的代码会新创建Spring容器对象并放入servletContext中，且键是“WebApplicationContext.ROOT”。

综上所述，初始化方法会把创建的Spring容器放到servletContext中了，这样这就解释Spring的生命周期为什么是servletContext范围了。

3. Web中的Spring容器还有一个很重要的知识，就是Spring容器会与当前线程绑定。

在ContextLoader类的301行，有这样的代码：

|  |
| --- |
| ***currentContextPerThread***.put(ccl, **this**.**context**); |

currentContextPerThread是一个Map<ClassLoader, WebApplicationContext>集合。集合的键ccl是当前线程的类加载器，值context就是当前创建好的Spring容器。

也就是说，把Spring容器绑定在了当前线程中。这有什么意义呢？这样就能在当前线程范围内，从任何地方都能拿到Spring容器。

例如因为业务需要，A类需要使用存放在Spring容器中的B类对象，但是A类并没有被Spring管理，这样如何拿到B对象呢？如果Spring容器没有绑定到当前线程，是无法拿到的。而现在，可通过以下方式拿到Spring容器及其中的对象：

|  |
| --- |
| *// 只要传入servletContext就能拿到Spring容器了* ApplicationContext c = WebApplicationContextUtils.*getWebApplicationContext*(servletContext); Person person = (Person) c.getBean(**"名称"**); |

其实getWebApplicationContext的代码如下：

|  |
| --- |
| **public static** WebApplicationContext getWebApplicationContext(ServletContext sc) {  **return** *getWebApplicationContext*(sc, WebApplicationContext.***ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE***); } |

也就是根据上面讲的键从servletContext中拿到对象。

## 3.2 关于Struts与Spring的整合

Struts与Spring的整合前面说过，插件机制做了很多工作。Tomcat启动时由于配置了Struts的过滤器，因此启动Struts后根据插件的配置文件，把Action的创建工作交由Spring管理了。

Spring容器启动时，由于DAO、Service层是单例的，则容器默认一启动就创建了DAO、Service的bean对象，同样，Service层的事务处理代理对象也会被创建完成。但此时Action层的对象是没有被创建的，因此Action类是多例的。

以下就是请求时执行的大致流程：

（1）用于请求一个action的URL；

（2）执行Struts2的核心过滤器；

（3）执行到DefaultActionInvocation的init方法中时，Struts会从Spring容器中实例化当前请求的Action（就是根据struts.xml中action标签的class属性匹配Spring容器中的bean的id来知道要创建的Action）。

## 3.3 OpenSessionInViewFilter解决Session关闭问题

在上述整合的DAO代码中，我们还并没有使用Hibernate的懒加载功能，现在，我们把BaseDaoImpl中的findById方法改为使用懒加载的load方法，即：

|  |
| --- |
| **public** T findById(Class<T> clazz, Serializable id) {  **return sessionFactory**.getCurrentSession().load(clazz, id); } |

此时，重新启动应用再访问“showStudent.action”这个URL，发现最终会出现以下错误：

|  |
| --- |
| org.hibernate.LazyInitializationException: could not initialize proxy - no Session |

“no Session”的原因是Hibernate的session已经被关闭了，因此当访问JSP页面时再想去取得懒加载数据时，此时session已经关闭了，因此出错。

我们在Service层使用了声明式事务处理后，session的有效范围只在Service中，在调用Service方法前后，Session分别被打开和关闭，因此如果使用了懒加载，在Service外要想取得懒加载数据就会报“no Session”错误。

那么怎么解决在action中获取懒加载数据的问题呢？Spring提供了“OpenSessionInViewFilter”这个在Servlet中使用的过滤器来解决这个问题。只要在web.xml中的Struts过滤器前加上“OpenSessionInViewFilter”过滤器即可解决这个问题：

|  |
| --- |
| <**filter**>  <**filter-name**>OpenSessionInView</**filter-name**>  <**filter-class**>org.springframework.orm.hibernate5.support.OpenSessionInViewFilter</**filter-class**> </**filter**> <**filter-mapping**>  <**filter-name**>OpenSessionInView</**filter-name**>  *<!-- 过滤路径和Struts2相同即可 -->* <**url-pattern**>\*.action</**url-pattern**> </**filter-mapping**> |

这样配置后再访问JSP页面就不会报错，能够拿到懒加载的数据。为什么在Struts过滤器前配置这样一个过滤器就能让JSP拿到懒加载数据，而不报session关闭的错误呢？这是因为OpenSessionInViewFilter在Struts2过滤器前面进行配置，当用于请求一个Action时，会先进入OpenSessionInViewFilter，该过滤器会提前拿到Spring容器中的sessionFactory并打开一个Session（Spring的监听器在Tomcat启动时就执行了，因此过滤器能拿到Spring容器），然后放行执行Struts2过滤器，Struts2中执行一系列的创建Action、执行拦截器、执行Action（包括Service、DAO）、执行结果集、返回到拦截器等操作，而最终，Struts2过滤器执行完毕后还会回到第一个过滤器OpenSessionInViewFilter中，该过滤器再关闭Session。

因此，由于先配置了OpenSessionInViewFilter过滤器，它把Session的生命周期延长了，因此执行JSP的数据渲染时，Session还是没有被关闭的，所以可正常获得懒加载数据，最后OpenSessionInViewFilter拦截器执行结束，JSP页面就被浏览器拿到显示在页面上了。

上述就是OpenSessionInViewFilter实现功能的原理，因此在开发时注意一定要将OpenSessionInViewFilter的配置写在Struts过滤器之前，否则是没有效果的。

## 3.4 基于Spring的单元测试

在上一节中，虽然解决了JSP中加载Session延迟数据的问题，但是在进行单元测试时，仍然是无法解决解决这个问题的。因为上述是利用Web中的过滤器解决问题的，提前开启Session且推迟关闭Session，因此在项目实际上线运行时，利用上述方法解决懒加载问题是没有问题的。

但是单元测试也很重要，一般会采用单元测试确保Service和DAO层的正确性，确保上层开发者调用是没有问题的。但是普通的单元测试如何成功进行单元测试呢？因为Service和DAO的测试时脱离Web环境的，测试时无法应用OpenSessionInViewFilter过滤器，如果像之前一样进行简单地单元测试，在Service必然会出现“no Session”错误（可自行验证），而在DAO层会出现以下错误（无法处理事务）：

|  |
| --- |
| HibernateException: Could not obtain transaction-synchronized Session for current thread |

对于基于Spring的项目，一般是利用Spring对JUnit测试框架的支持，进行如下的单元测试方法，步骤如下：

（1）在项目中引入Spring对测试的支持包“spring-test-5.0.2.RELEASE.jar”，且引入JUnit的相关包；

（2）创建一个与src目录同级的目录，例如就叫“test”目录，并设置为源码目录。以后该目录就专门存放测试类。（这步不是必须的，但一般会这么干，项目实际代码和测试代码分开比较清晰。注意一定要设置本目录为源码目录，这样其字节码文件也会在classpath下）

（3）新建包（例如叫com.ssh.test.service），创建测试类（例如StudentServiceTest）。编写测试类中要注意以下几点：

1. 在类上使用@RunWith注解指定Spring对JUnit的支持类；

2. 在类上使用@ContextConfiguration注解指定Spring的XML配置文件的位置。

3. 如果要测试Service类中方法，则应在本测试类中注入对应的Service对象属性，当然如果要测试DAO类中方法，则应在本测试类中注入对应的DAO对象属性。

4. 由于单独测试Service和DAO都需要事务的支持（获取Session和控制事务等），因此在测试的方法上也应该加上@Transactional注解，当然在类上使用@Transactional注解也可，这样类中方法都会用到事务。

5. 基于Spring进行测试时，对数据库的操作默认都会进行回滚，即即使测试成功后，数据库中数据也是不发生改变的，因为测试完会自动回滚数据。如果不想回滚数据，而想在测试后看到数据库中的效果，则应该使用@Rollback注解指示不回滚。

代码示例：

|  |
| --- |
| **package** com.ssh.test;  **import** com.ssh.dao.IStudentDao; **import** com.ssh.domain.Student; **import** com.ssh.service.IStudentService; **import** org.junit.Test; **import** org.junit.runner.RunWith; **import** org.springframework.test.annotation.Rollback; **import** org.springframework.test.context.ContextConfiguration; **import** org.springframework.test.context.junit4.SpringJUnit4ClassRunner; **import** org.springframework.transaction.annotation.Transactional; **import** javax.annotation.Resource; **import** java.util.Date;  *// @RunWith注解指示支持类* @RunWith(SpringJUnit4ClassRunner.**class**) *// @ContextConfiguration注解加载Spring的配置文件* @ContextConfiguration(locations={**"classpath:applicationContext.xml"**}) **public class** StudentServiceTest {   *// 注入需要的属性  // 本测试类既要测试Service，又要测试DAO，一起注入了（实际中应该分开）。* @Resource(name = **"studentService"**)  **private** IStudentService **studentService**;  @Resource(name = **"studentDao"**)  **private** IStudentDao **studentDao**;   @Test  @Transactional(readOnly = **true**) *// 使用@Transactional* **public void** testService() {  *// 测试service中的懒加载数据* Student student = **studentService**.findById(1);  System.***out***.println(student);  }   @Test  @Transactional(readOnly = **true**)  **public void** testDao() {  *// 测试DAO中的懒加载，方法上也要使用@Transactional注解* Student student = **studentDao**.findById(Student.**class**, 1);  System.***out***.println(student.getName());  }   @Test  @Transactional(readOnly = **false**)  @Rollback(**false**) *// 不回滚数据* **public void** testDaoUpdqateData() {  *// 添加数据，因此@Transactional设置为false。  // 为了看到测试的数据，设置了不回滚，否则看不到数据* **studentDao**.add(**new** Student(2, **"李四"**, **new** Date()));  } } |

利用上述方法进行测试不会出现懒加载问题。在实际中可以采取这样的测试方法。

那么有人又要提出疑问了，那么如果不是WEB项目，也不是测试代码，如果上层要用到懒加载数据怎么办呢？那这里我暂时没有得到很好的解决办法。因为不是WEB项目的话，没法使用过滤器，那你使用怎么控制在哪里获得Session，并在哪里关闭Session呢？目前我还没有好办法。除非你对Spring其中的原理很了解，可以自己写类来进行控制，否则你只能放弃使用懒加载了。

# 4. 最后说明

本整合的项目示例放到了如下网站上：

https://gitee.com/zhang13690/sshdemo/tree/v1/

读者可参考。最后，希望读者能用SSH框架做一个自己感兴趣的小项目，以提高自己解决问题的经验和能力，这非常重要。同时还应不断自学Spring相关知识。