尚学堂 Spring 学习笔记

目录

一、锲子.....2
二、环境.....2
三、环境搭建.....2
四、关键技术.....2
Ioc 控制反转.....2
AOP（面向切面编程）.....3
spring 对 AOP 的支持（采用 Annotation 的方式）.....3
spring 对 AOP 的支持（采用配置文件的方式）.....5
spring 对 AOP 的支持（关于接口）.....5
spring 对 AOP 的支持（关于接口）二.....6
Autowrire（自动装配）.....6
根据名称自动装配.....6
根据类型自动装配.....6
Injection（依赖注入）.....7
Proxy（代理）.....9
静态代理.....9
动态代理.....10
spring Bean 的作用域.....11
五、整合.....11
Spring + Hibernate.....11
采用编程式事务.....11
采用声明式事务.....12
Spring + Struts.....14
spring+struts 的集成(第一种集成方案).....14
spring+struts 的集成(第二种集成方案).....16
Spring + Struts + Hibernate（SSH）.....17
六、写在最后.....17

一、楔子

在前一段时间内学习了一些 Spring 的知识，感觉还是需要整理一下自己的东西，不然我感觉很是容易遗忘。

二、环境

本次学习用到了一些软硬件环境如下：

1. MyEclipse Enterprise Workbench 7.0 Milestone-1（MyEclipse 7.0 M1）
2. Spring 2.5
3. Struts 1.2
4. Hibernate 3.2
5. 其他（SVN 等）

三、环境搭建

1、spring依赖库

* SPRING_HOME/dist/spring.jar
* SPRING_HOME/lib/jakarta-commons/commons-logging.jar
* SPRING_HOME/lib/log4j/log4j-1.2.14.jar

2、拷贝spring配置文件(applicationContext.xml)到src下

3、拷贝log4j配置文件（log4j.properties）到src下

提示：上面的一些jar包如果在现在的MyEclipse环境里可以通过添加对Spring的支持自动添加，上面的一些配置文件都可以再Spring官方下载的Spring包中的例子程序中找到相应的一些文件，修改后就可以使用

四、关键技术

Ioc 控制反转

spring Ioc容器的关键点：

* 必须将被管理的对象定义到spring配置文件中
* 必须定义构造函数或 setter 方法，让 spring 将对象注入过来

```
<bean id="唯一标识" class="Ioc容器管理的需要注入的类 ">

        <!-- 构造方法注入 -->
```

```
        <constructor-arg …/>
        <!--set 方法-->
         <property name="set方法的属性" …/>
</bean>
```

后面通过读取配置文件，并新建工厂，通过工厂来获得 bean

```
BeanFactory factory = new
ClassPathXmlApplicationContext("applicationContext.xml");
UserManager userManager = (UserManager)factory.getBean("XX");
```

AOP（面向切面编程）

spring 对 AOP 的支持（采用 Annotation 的方式）

1、spring依赖库
    * SPRING_HOME/dist/spring.jar
    * SPRING_HOME/lib/jakarta-commons/commons-logging.jar
    * SPRING_HOME/lib/log4j/log4j-1.2.14.jar
    * SPRING_HOME/lib/aspectj/*.jar

2、采用Aspect定义切面

2、在Aspect定义Pointcut和Advice

4、启用AspectJ对Annotation的支持并且将Aspect类和目标对象配置到Ioc容器中

注意：在这种方法定义中，切入点的方法是不被执行的，它存在的目的仅仅是为了重用切入点即Advice中通过方法名引用这个切人点

AOP一些关键技术：
    * Cross cutting concern
    * Aspect
    * Advice
    * Pointcut
    * Joinpoint
    * Weave
    * Target Object
    * Proxy
    * Introduction

注意使用 AOP 的时候一定要添加相应的类库（jar 包）

建一个 Annotations 的类，然后通过添加 Spring 对 AOP 的支持，也就是在 Spring 的配置文件中加入
<aop:aspectj-autoproxy/>
就可以了

代码如下

```
package com.bjsxt.spring;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

/**
 * 定义 Aspect
 * @author Administrator
 *
 */
@Aspect
public class SecurityHandler {

    /**
     * 定义 Pointcut,Pointcut 的名称就是 allAddMethod，此方法不能有返回值和参数，该方法只是一个
     * 标识
     *
     * Pointcut 的内容是一个表达式，描述那些对象的那些方法（订阅 Joinpoint）
     */
    @Pointcut("execution(* add*(..)) || execution(* del*(..))")
    private void allAddMethod(){};

    /**
     * 定义 Advice，标识在那个切入点何处织入此方法
     */
    @Before("allAddMethod()")
    private void checkSecurity() {
        System.out.println("----------checkSecurity()----------");
    }

}
```

客户端代码不变

spring 对 AOP 的支持（采用配置文件的方式）

1、spring依赖库
* SPRING_HOME/dist/spring.jar
* SPRING_HOME/lib/jakarta-commons/commons-logging.jar
* SPRING_HOME/lib/log4j/log4j-1.2.14.jar
* SPRING_HOME/lib/aspectj/*.jar

2、配置如下

```
<aop:config>
    <aop:aspect id="security" ref="securityHandler">
        <aop:pointcut id="allAddMethod" expression="execution(*
com.bjsxt.spring.UserManagerImpl.add*(..))"/>
        <aop:before method="checkSecurity"
pointcut-ref="allAddMethod"/>
    </aop:aspect>
</aop:config>
```

spring 对 AOP 的支持（关于接口）

spring对AOP的支持

Aspect默认情况下不用实现接口，但对于目标对象（UserManagerImpl.java），在默认情况下必须实现接口
如果没有实现接口必须引入CGLIB库

我们可以通过Advice中添加一个JoinPoint参数，这个值会由spring自动传入，从JoinPoint中可以取得
参数值、方法名等等

代码

```
package com.bjsxt.spring;

import org.aspectj.lang.JoinPoint;

public class SecurityHandler {

    private void checkSecurity(JoinPoint joinPoint) {
        Object[] args = joinPoint.getArgs();
        for (int i=0; i<args.length; i++) {
            System.out.println(args[i]);
        }
```

```
        System.out.println(joinPoint.getSignature().getName());

    System.out.println("----------checkSecurity()---------------");
    }

}
```

客户端代码不变

spring 对 AOP 的支持（关于接口）二

1、如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP
2、如果目标对象实现了接口，可以强制使用CGLIB实现AOP
3、如果目标对象没有实现了接口，必须采用CGLIB库，spring会自动在JDK动态代理和CGLIB之间转换

如何强制使用CGLIB实现AOP?

* 添加CGLIB库，SPRING_HOME/cglib/*.jar
* 在spring配置文件中加入<aop:aspectj-autoproxy proxy-target-class="true"/>

JDK动态代理和CGLIB字节码生成的区别?

* JDK动态代理只能对实现了接口的类生成代理，而不能针对类
* CGLIB是针对类实现代理，主要是对指定的类生成一个子类，覆盖其中的方法
  因为是继承，所以该类或方法最好不要声明成final

Autowrire（自动装配）

根据名称自动装配

在 Spring 的配置文件头中加入 default-autowire="byName"
其中 bean 里面的 id 一定与你的 bean 类名字一样，不然找不到并装配不了

根据类型自动装配

在 Spring 的配置文件头中加入 default-autowire="byType"，Spring 默认装配方式
其中 bean 里面的 id 可以与你的 bean 类名字不一样，可以通过 class 来查找你需要注入的属性

Injection（依赖注入）

spring的普通属性注入

什么是属性编辑器，作用？

* 自定义属性编辑器，spring配置文件中的字符串转换成相应的对象进行注入

spring已经有内置的属性编辑器，我们可以根据需求自己定义属性编辑器

* 如何定义属性编辑器？
    * 继承PropertyEditorSupport类，覆写setAsText()方法

```
package com.bjsxt.spring;

import java.beans.PropertyEditorSupport;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * java.util.Date属性编辑器
 * @author Administrator
 *
 */
public class UtilDatePropertyEditor extends PropertyEditorSupport {

    private String format="yyyy-MM-dd";

    @Override
    public void setAsText(String text) throws IllegalArgumentException
{
        SimpleDateFormat sdf = new SimpleDateFormat(format);
        try {
            Date d = sdf.parse(text);
            this.setValue(d);
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }

    public void setFormat(String format) {
        this.format = format;
    }
```

```
}
```

* 将属性编辑器注册到spring中

```
<!-- 定义属性编辑器 -->
    <bean id="customEditorConfigurer"
class="org.springframework.beans.factory.config.CustomEditorConfigure
r">
        <property name="customEditors">
            <map>
                <entry key="java.util.Date">
                    <bean
class="com.bjsxt.spring.UtilDatePropertyEditor">
                        <property name="format" value="yyyy-MM-dd"/>
                    </bean>
                </entry>
            </map>
        </property>
    </bean>
```

依赖对象的注入方式，可以采用：

* ref属性
* <ref>标签
* 内部<bean>来定义

如何将公共的注入定义描述出来？

* 通过<bean>标签定义公共的属性，指定abstract=true
* 具有相同属性的类在<bean>标签中指定其parent属性

```
<bean id="beanAbstract" abstract="true">
        <property name="id" value="1000"/>
        <property name="name" value="Jack"/>
   </bean>

   <bean id="bean3" class="com.bjsxt.spring.Bean3"
parent="beanAbstract">
<!--        <property name="name" value="Tom"/>-->
        <property name="password" value="123"/>
   </bean>

   <bean id="bean4" class="com.bjsxt.spring.Bean4"
parent="beanAbstract"/>
```

Proxy（代理）

静态代理

静态代理只需写一个静态代理类就可以了

```
package com.bjsxt.spring;

public class UserManagerImplProxy implements UserManager {

    private UserManager userManager;

    public UserManagerImplProxy(UserManager userManager) {
        this.userManager = userManager;
    }

    public void addUser(String username, String password) {
        checkSecurity();
        this.userManager.addUser(username, password);
    }

    public void deleteUser(int id) {
        checkSecurity();
        this.userManager.deleteUser(id);
    }

    public String findUserById(int id) {
        return null;
    }

    public void modifyUser(int id, String username, String password) {
    }

    private void checkSecurity() {
    System.out.println("----------checkSecurity()---------------");
}

}
```

客户端代码

```
UserManager userManager = new UserManagerImplProxy(new UserManagerImpl());
userManager.addUser("张三", "123");
```

动态代理

动态代理的话就要写一个 Handler，通过 Handler 来生成管理类的代理

```
package com.bjsxt.spring;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class SecurityHandler implements InvocationHandler {

	private Object targetObject;

	public Object newProxy(Object targetObject) {
		this.targetObject = targetObject;
		return Proxy.newProxyInstance(targetObject.getClass().getClassLoader(),
						targetObject.getClass().getInterfaces(),
						this);

	}

	public Object invoke(Object proxy, Method method, Object[] args)
			throws Throwable {
		checkSecurity();
		Object ret = null;
		try {
			ret = method.invoke(this.targetObject, args);
		}catch(Exception e) {
			e.printStackTrace();
			throw new java.lang.RuntimeException(e);
		}
		return ret;
	}

	private void checkSecurity() {
		System.out.println("----------checkSecurity()----------");
	}

}
```

客户端代码

```
SecurityHandler handler = new SecurityHandler();
UserManager userManager = (UserManager)handler.newProxy(new UserManagerImpl());

userManager.addUser("张三", "123");
```

spring Bean 的作用域

scope可以取值：
* singleton:每次调用getBean的时候返回相同的实例
* prototype:每次调用 getBean 的时候返回不同的实例

```
<bean id="XX " class=" " scope="prototype"/>
或者
<bean id="XX " class=" " scope=" singleton "/>
```

五、整合

Spring + Hibernate

采用编程式事务

1、getCurrentSession()与openSession()的区别？
* 采用getCurrentSession()创建的session会绑定到当前线程中，而采用openSession()
  创建的session则不会
* 采用getCurrentSession()创建的session在commit或rollback时会自动关闭，而采用openSession()
  创建的session必须手动关闭

2、使用getCurrentSession()需要在hibernate.cfg.xml文件中加入如下配置：
* 如果使用的是本地事务（jdbc事务）
  `<property name="hibernate.current_session_context_class">thread</property>`
* 如果使用的是全局事务（jta事务）
  `<property name="hibernate.current_session_context_class">jta</property>`

如

```
<!--    使用getcurrentsession方法的配置    -->
<!--
使用getCurrentSession()需要在hibernate.cfg.xml文件中加入如下配置：
    * 如果使用的是本地事务（jdbc事务）
    <property
name="hibernate.current_session_context_class">thread</property>
    * 如果使用的是全局事务（jta事务）
    <property
name="hibernate.current_session_context_class">jta</property>
-->
```

采用声明式事务

1、声明式事务配置

* 配置SessionFactory
* 配置事务管理器
* 事务的传播特性
* 那些类那些方法使用事务

2、编写业务逻辑方法

* 继承HibernateDaoSupport类，使用HibernateTemplate来持久化，HibernateTemplate是
  Hibernate Session的轻量级封装
* 默认情况下运行期异常才会回滚（包括继承了RuntimeException子类），普通异常是不会回滚的
* 编写业务逻辑方法时，最好将异常一直向上抛出，在表示层（struts）处理
* 关于事务边界的设置，通常设置到业务层(Manager)，不要添加到Dao上

3、了解事务的几种传播特性

1. PROPAGATION_REQUIRED：如果存在一个事务，则支持当前事务。如果没有事务则开启
2. PROPAGATION_SUPPORTS：如果存在一个事务，支持当前事务。如果没有事务，则非事务的执行
3. PROPAGATION_MANDATORY：如果已经存在一个事务，支持当前事务。如果没有一个活动的事务，则抛出异常。
4. PROPAGATION_REQUIRES_NEW：总是开启一个新的事务。如果一个事务已经存在，则将这个存在的事务挂起。
5. PROPAGATION_NOT_SUPPORTED：总是非事务地执行，并挂起任何存在的事务。
6. PROPAGATION_NEVER：总是非事务地执行，如果存在一个活动事务，则抛出异常
7. PROPAGATION_NESTED：如果一个活动的事务存在，则运行在一个嵌套的事务中．如果没有活动事务，
   则按TransactionDefinition.PROPAGATION_REQUIRED 属性执行

4、Spring事务的隔离级别

1. ISOLATION_DEFAULT: 这是一个PlatfromTransactionManager默认的隔离级别，使用数据库默认的事务隔离级别.

另外四个与JDBC的隔离级别相对应

2. ISOLATION_READ_UNCOMMITTED: 这是事务最低的隔离级别，它充许令外一个事务可以看到这个事务未提交的数据。

这种隔离级别会产生脏读，不可重复读和幻像读。

3. ISOLATION_READ_COMMITTED: 保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的数据

4. ISOLATION_REPEATABLE_READ: 这种事务隔离级别可以防止脏读，不可重复读。但是可能出现幻像读。

它除了保证一个事务不能读取另一个事务未提交的数据外，还保证了避免下面的情况产生(不可重复读)。

5. ISOLATION_SERIALIZABLE 这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。

除了防止脏读，不可重复读外，还避免了幻像读。

配置声明式事务

```
<?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:aop="http://www.springframework.org/schema/aop"
        xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
         http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd
         http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">
    <!-- 配置sessionFactory -->
    <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
       <property name="configLocation">
           <value>classpath:hibernate.cfg.xml</value>
       </property>
    </bean>

    <!-- 配置事务管理器 -->
    <bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager
">
```

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

    <!-- 配置事务的传播特性 -->
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="add*" propagation="REQUIRED"/>
            <tx:method name="del*" propagation="REQUIRED"/>
            <tx:method name="modify*" propagation="REQUIRED"/>
            <tx:method name="*" read-only="true"/>
        </tx:attributes>
    </tx:advice>


    <!-- 那些类的哪些方法参与事务 -->
    <aop:config>
        <aop:pointcut id="allManagerMethod" expression="execution(*
com.bjsxt.usermgr.manager.*.*(..))"/>
        <aop:advisor pointcut-ref="allManagerMethod"
advice-ref="txAdvice"/>
    </aop:config>
</beans>
```

通过他来获得 seesion factory

```
<bean id="logManager"
class="com.bjsxt.usermgr.manager.LogManagerImpl">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>
```

Spring + Struts

spring+struts 的集成(第一种集成方案)

原理：在Action中取得BeanFactory对象，然后通过BeanFactory获取业务逻辑对象

1、spring和struts依赖库配置

* 配置struts

--拷贝struts类库和jstl类库

--修改web.xml文件来配置ActionServlet

--提供struts-config.xml文件

--提供国际化资源文件
* 配置spring
--拷贝spring类库
--提供spring配置文件

2、在struts的Action中调用如下代码取得BeanFactory

```
BeanFactory factory =
WebApplicationContextUtils.getRequiredWebApplicationContext(request.getSession().getServletContext());
```

3、通过BeanFactory取得业务对象，调用业务逻辑方法

Action里的

```
////        1.直接使用，没有使用spring
//      UserManager userManager = new UserManagerImpl();
//      userManager.login(laf.getUsername(), laf.getPassword());

////        2.使用spring的重量级的工厂
//      BeanFactory factory = new
ClassPathXmlApplicationContext("applicationContext-beans.xml");
//      UserManager userManager =
(UserManager)factory.getBean("userManager");
//      userManager.login(laf.getUsername(), laf.getPassword());

//          3.使用web.xml来配置listener来读取配置文件来创建工厂
        BeanFactory factory =
WebApplicationContextUtils.getRequiredWebApplicationContext(request.getSession().getServletContext());

        //ApplicationContext继承beanfactory，所以不用强制转化，也可以使用下面的活动工厂
        //ApplicationContext pc =
WebApplicationContextUtils.getRequiredWebApplicationContext(request.getSession().getServletContext());
        UserManager userManager =
(UserManager)factory.getBean("userManager");
        userManager.login(laf.getUsername(), laf.getPassword());

        return mapping.findForward("success");
```

web.xml里的配置（添加部分）

```
  <context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>classpath*:applicationContext-*.xml</param-value>
```

```
  </context-param>

    <listener>

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

spring+struts 的集成(第二种集成方案)

原理：将业务逻辑对象通过spring注入到Action中，从而避免了在Action类中的直接代码查询

1、spring和struts依赖库配置
    * 配置struts
        --拷贝struts类库和jstl类库
        --修改web.xml文件来配置ActionServlet
        --提供struts-config.xml文件
        --提供国际化资源文件
    * 配置spring
        --拷贝spring类库
        --提供spring配置文件
2、因为Action需要调用业务逻辑方法，所以需要在Action中提供setter方法，让spring将业务逻辑对象注入过来

3、在struts-config.xml文件中配置Action
    * <action>标签中的type属性需要修改为org.springframework.web.struts.DelegatingActionProxy
     DelegatingActionProxy是一个Action，主要作用是取得BeanFactory，然后根据<action>中的path属性值
     到IoC容器中取得本次请求对应的Action

4、在spring配置文件中需要定义struts的Action,如：

```
    <bean name="/login" class="com.bjsxt.usermgr.actions.LoginAction" scope="prototype">
        <property name="userManager" ref="userManager"/>
    </bean>
```

    * 必须使用name属性,name属性值必须和struts-config.xml文件中<action>标签的path属性值一致
    * 必须注入业务逻辑对象
    * 建议将scope设置为prototype,这样就避免了struts Action的线程安全问题

```
    <bean name="/login" class="com.bjsxt.usermgr.actions.LoginAction"
scope="prototype">
        <property name="userManager" ref="userManager"/>
    </bean>

<!--type不是以前的类了，是代理类了                -->
        <action path="/login"

    type="org.springframework.web.struts.DelegatingActionProxy"
                name="loginForm"
                scope="request"
        >
            <forward name="success" path="/success.jsp"/>
        </action>
    </action-mappings>
```

Spring + Struts + Hibernate（SSH）

整合上面的 SH + SS，就可以了，本人这部分也不是很熟练，先就不写了

六、写在最后

刚学了一点关于 Spring 的东西，里面也仅是总结，并没有其他一些自己的思想，总结出来为了我以后跟好地学习。

大家如果有什么好的思想，有空加我 QQ（506817493），很期待与你交流，谢谢。

木子写于 2009 年 6 月 2 日