

Spring_day01 总结

今日任务

- 使用 Spring 完成对客户的保存操作

教学导航

教学目标	
教学方法	案例驱动法

案例一使用 Spring 的 IOC 完成保存客户的操作：

1.1 案例需求

1.1.1 需求概述

CRM 系统中客户信息管理模块功能包括：

- 新增客户信息
- 客户信息查询
- 修改客户信息
- 删除客户信息

本功能要实现新增客户，页面如下：

当前位置：客户管理 > 添加客户

客户名称：

客户级别：

信息来源：

联系人：

固定电话：

移动电话：

联系地址：

邮政编码：

客户传真：

客户网址：

保存

1.2 相关知识点

1.1.1 Spring 的概述:

1.2.1.1 什么是 Spring :

spring (由Rod Johnson创建的一个开源框架)

编辑

Spring是一个开源框架，Spring是于2003年兴起的一个轻量级的Java开发框架，由Rod Johnson创建。简单来说，Spring是一个分层的JavaSE/EEfull-stack(一站式)轻量级开源框架。

Spring是一个开源框架，Spring是于2003年兴起的一个轻量级的Java开发框架，由[Rod Johnson](#)在其著作 *Expert One-On-One J2EE Development and Design* 中阐述的部分理念和原型衍生而来。它是为了解决企业应用开发的复杂性而创建的。框架的主要优势之一就是其分层架构，分层架构允许使用者选择使用哪一个组件，同时为 [J2EE](#) 应用程序开发提供集成的框架。Spring使用基本的 [JavaBean](#) 来完成以前只可能由 EJB 完成的事情。然而，Spring的用途不仅限于[服务器端](#)的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益。Spring的核心是[控制反转](#) ([IoC](#)) 和面向切面 ([AOP](#))。简单来说，**Spring 是一个分层的 JavaSE/EEfull-stack(一站式) 轻量级开源框架。**

EE 开发分成三层结构:

- * WEB 层:Spring MVC.
- * 业务层:Bean 管理:(IOC)
- * 持久层:Spring 的 JDBC 模板.ORM 模板用于整合其他的持久层框架.

Expert One-to-One J2EE Design and Development	:J2EE 的设计和开发:(2002.EJB)
Expert One-to-One J2EE Development without EJB	:J2EE 不使用 EJB 的开发.

1.2.1.2 为什么学习 Spring:

方便解耦，简化开发

Spring 就是一个大工厂，可以将所有对象创建和依赖关系维护，交给 Spring 管理
AOP 编程的支持

Spring 提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能
声明式事务的支持

只需要通过配置就可以完成对事务的管理，而无需手动编程
方便程序的测试

Spring 对 Junit4 支持，可以通过注解方便的测试 Spring 程序
方便集成各种优秀框架

Spring 不排斥各种优秀的开源框架，其内部提供了对各种优秀框架（如：Struts、Hibernate、MyBatis、Quartz 等）的直接支持

降低 JavaEE API 的使用难度

Spring 对 JavaEE 开发中非常难用的一些 API（JDBC、JavaMail、远程调用等），都提供了封装，使这些 API 应用难度大大降低

1.2.1.3 Spring 的版本:

Spring 3.X 和 Spring 4.X

1.2.2 Spring 的入门案例:(IOC)

1.2.2.1 IOC 的底层实现原理

Spring 的 IOC 底层实现原理:

传统方式开发:

```
UserDao userDao = new UserDao();
```

↓ 面向接口编程.

```
UserDao userDao = new UserDaoImpl();
```

↓ 底层实现类的切换.

```
* UserDaoImpl -- 使用JDBC实现  
* UserDaoHibernateImpl -- 使用Hibernate实现
```

切换底层的实现类 需要修改程序的源代码的.

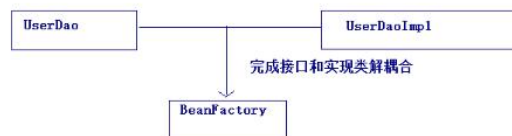
程序设计有一个原则:OCP原则,open-close原则.

* 程序设计的时候 尽量做到对程序的扩展是open的,对修改源代码是close的.

* 好的程序设计 尽量少修改源码的基础上对程序进行扩展.

↓ 完成接口和实现类的解耦合的操作

工厂模式:



完成接口和实现类解耦合

```
BeanFactory{  
    public static UserDao getUserDao(){  
        return new UserDaoImpl();  
        UserDaoHibernateImpl()  
    }  
    public static CustomerDao getCustomerDao(){  
        return new CustomerDaoImpl();  
    }  
}
```

接口和工厂有耦合

```
UserDao userDao = BeanFactory.getUserDao();
```

工厂+反射+配置文件

```
<bean id="userDao" class="cn.itcast.dao.UserDaoImpl"/>  
// 解析XML  
BeanFactory{  
    public static Object getBean(String id){  
        Class clazz = Class.forName();  
        return clazz.newInstance();  
    }  
}
```

```
UserDao userDao = BeanFactory.getBean("userDao");
```

IOC: Inversion of Control 控制反转. 指的是 对象的创建权反转(交给)给 Spring.

作用是实现了程序的解耦合.

1.2.2.2 步骤一:下载 Spring 的开发包:

官网: <http://spring.io/>

下 载

地

址

:

<http://repo.springsource.org/libs-release-local/org/springframework/spring>

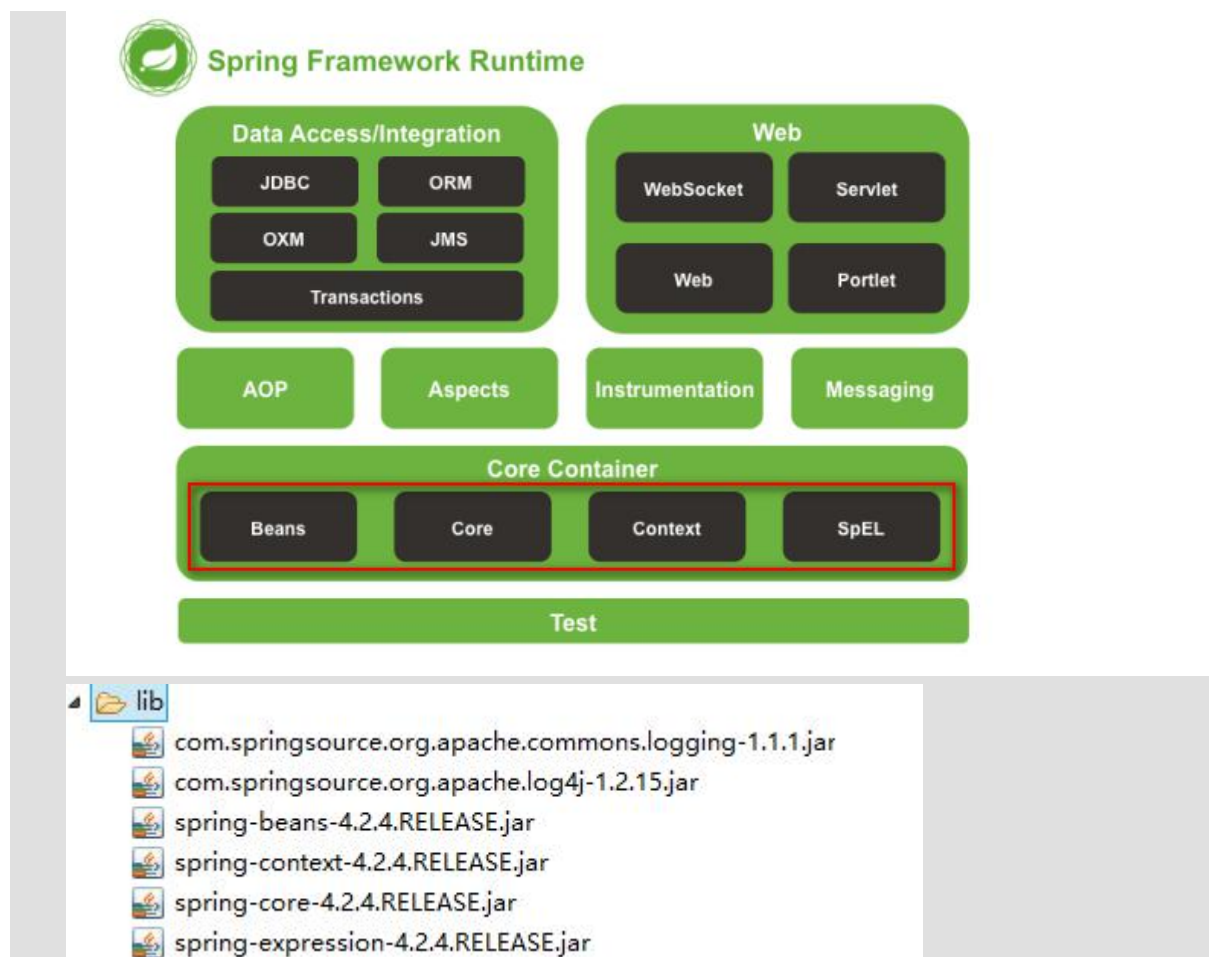
解

压: (Spring 目录结构:)

* docs :API 和开发规范.

- * libs :jar 包和源码.
- * schema :约束.

1.2.2.3 步骤二:创建 web 项目,引入 Spring 的开发包:



1.2.2.4 步骤三:引入相关配置文件:

```
log4j.properties
applicationContext.xml
引入约束:
spring-framework-4.2.4.RELEASE\docs\spring-framework-reference\html\xsd-configuration.html

<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">
```

```
</beans>
```

1.2.2.5 步骤四:编写相关的类:

```
public interface UserDao {

    public void sayHello();

}

public class UserDaoImpl implements UserDao {

    @Override
    public void sayHello() {
        System.out.println("Hello Spring...");
    }

}
```

1.2.2.6 步骤五:完成配置:

```
<!-- Spring 的入门案例===== -->
<bean id="userDao" class="cn.itcast.spring.demol.UserDaoImpl"></bean>
```

1.2.2.7 步骤六:编写测试程序:

```
@Test
// Spring 的方式:
public void demo2() {
    // 创建 Spring 的工厂类:
    ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
    // 通过工厂解析 XML 获取 Bean 的实例.
    UserDao userDao = (UserDao) applicationContext.getBean("userDao");
    userDao.sayHello();
}
```

1.2.2.8 IOC 和 DI:

IOC :控制反转,将对象的创建权交给了 Spring.

DI :Dependency Injection 依赖注入.需要有 IOC 的环境,Spring 创建这个类的过程中,Spring 将类的依

赖的属性设置进去.

1.2.3 Spring 中的工厂:

1.2.3.1 ApplicationContext:

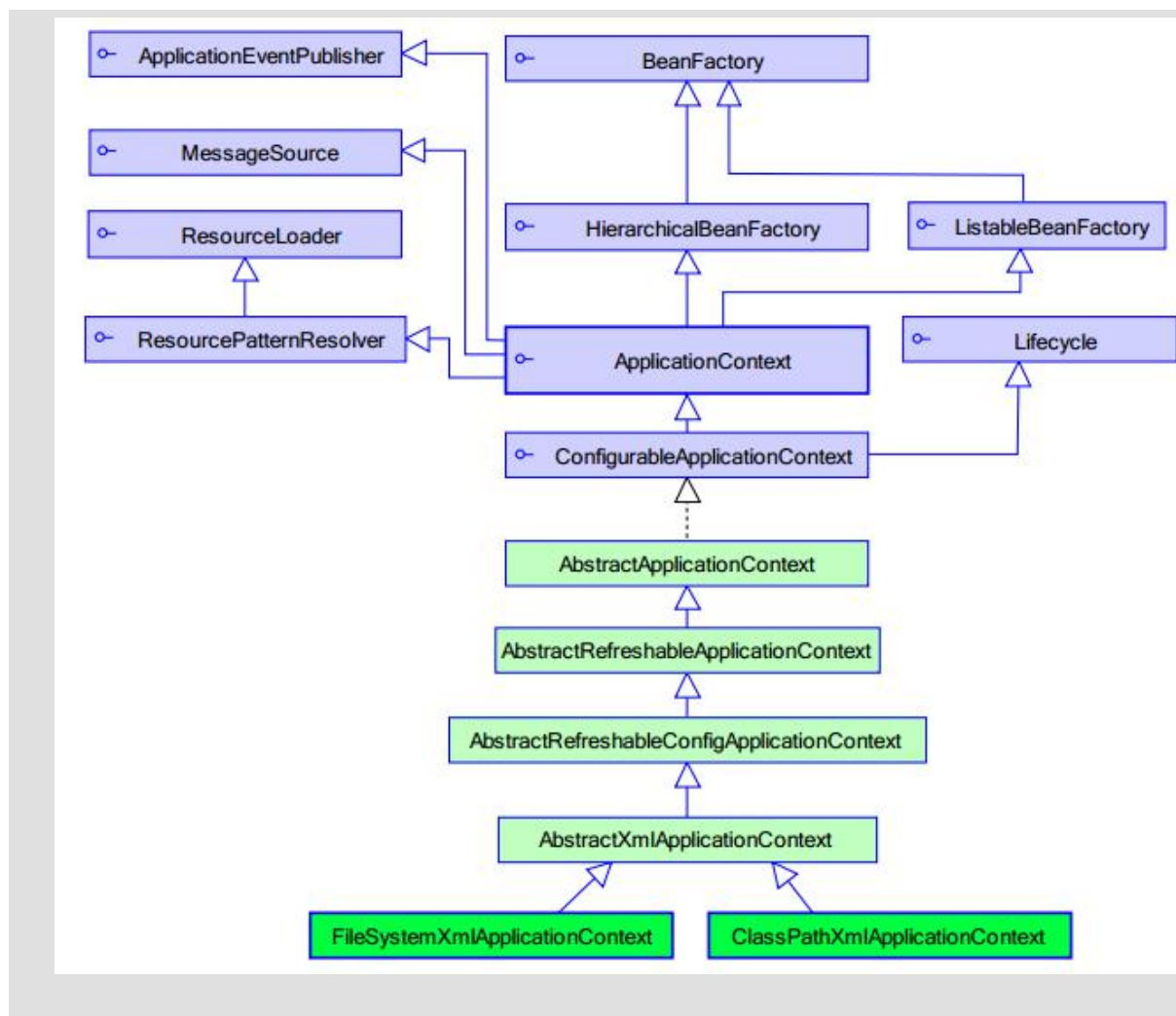
ApplicatioContext 接口有两个实现类:



`ClassPathXmlApplicationContext` :加载类路径下 Spring 的配置文件.

`FileSystemXmlApplicationContext` :加载本地磁盘下 Spring 的配置文件.

1.2.3.2 BeanFactory:



1.2.3.3 BeanFactory 和 ApplicationContext 的区别:

BeanFactory :是在 getBean 的时候才会生成类的实例。

ApplicationContext :在加载 applicationContext.xml 时候就会创建。

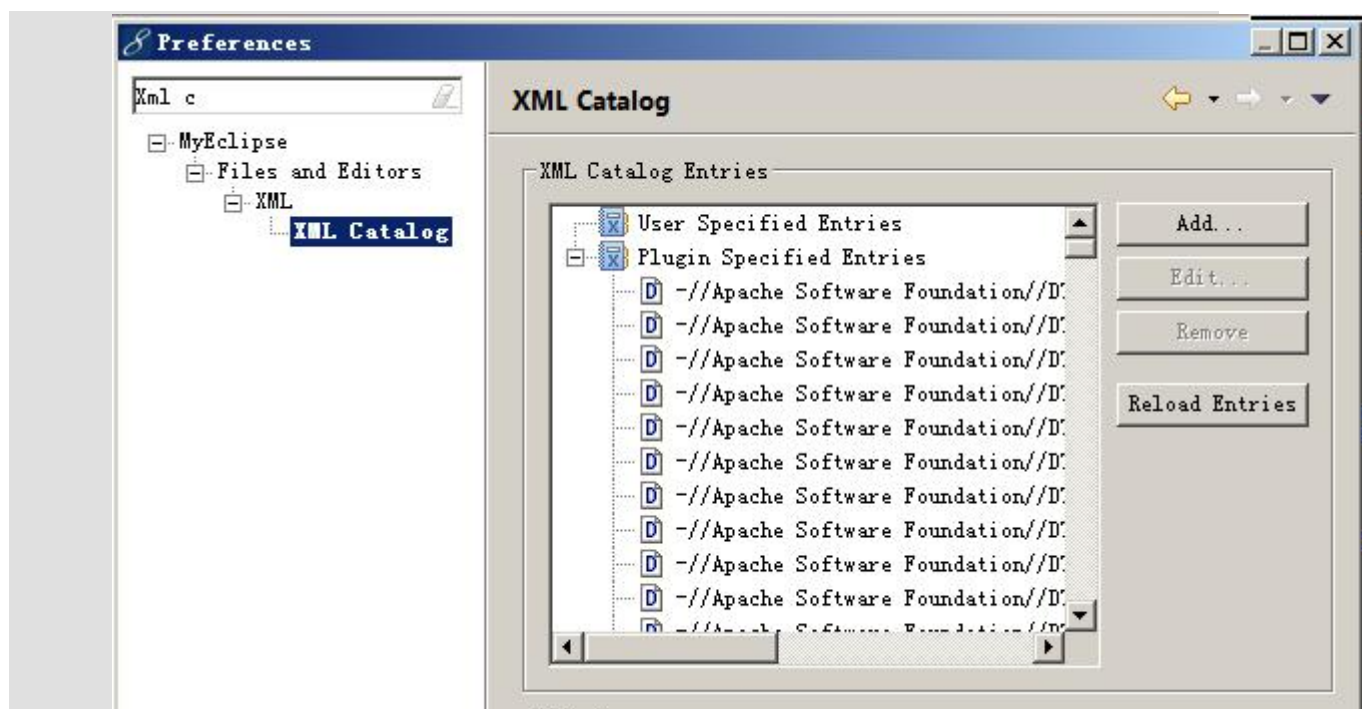
1.2.4 配置 STS 的 XML 的提示:

1.2.4.1 Spring 配置文件中提示的配置

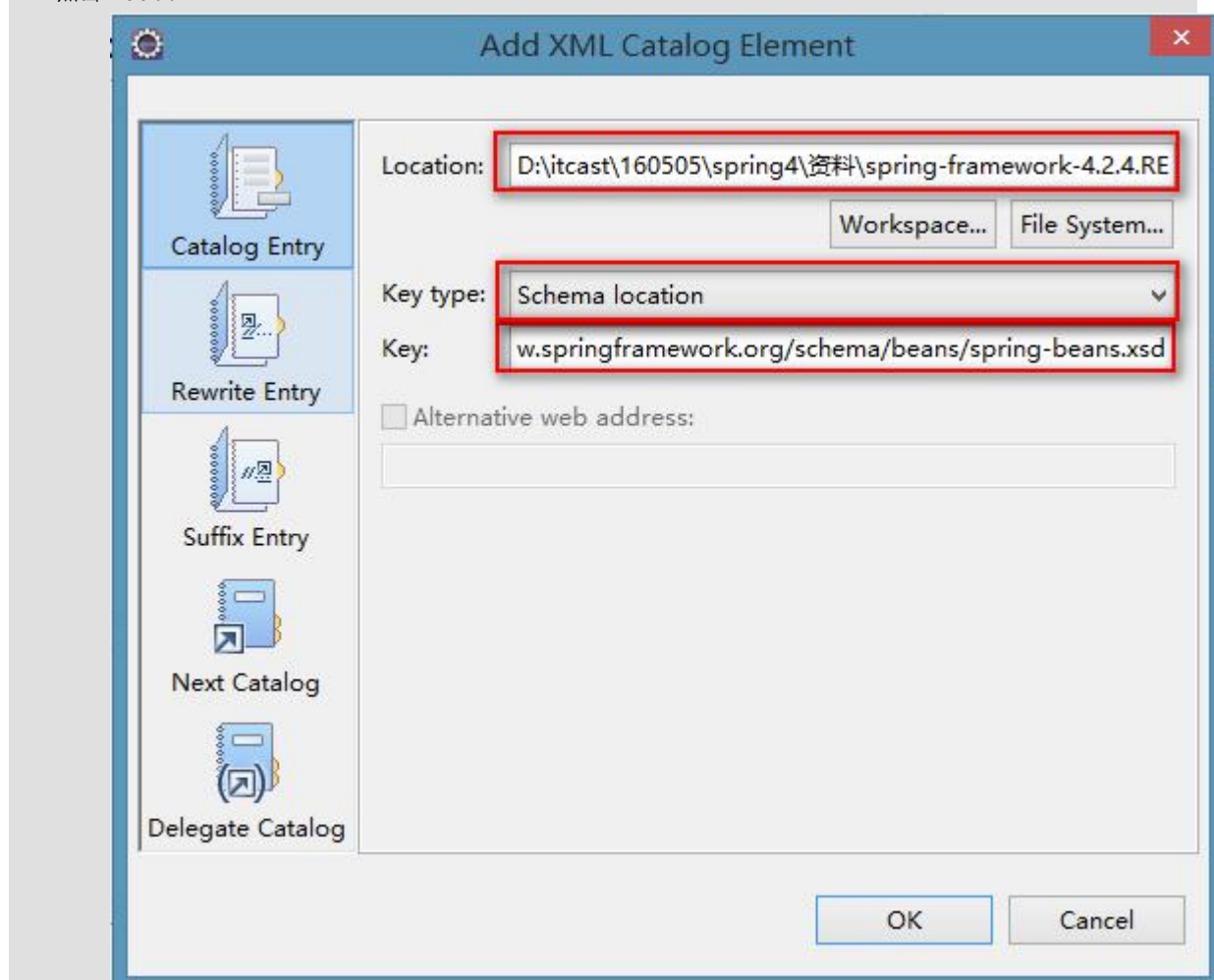
复制路径:

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

查找 XML Catalog:



点击 Add...



1.2.5 Spring 的相关配置:

1.2.5.1 id 属性和 name 属性标签的配置

id :Bean 起个名字. 在约束中采用 ID 的约束:唯一.必须以字母开始,可以使用字母、数字、连字符、下划线、句点、冒号 id:不能出现特殊字符.

```
<bean id="bookAction">
```

name:Bean 起个名字. 没有采用 ID 的约束. name:出现特殊字符.如果<bean>没有 id 的话, name 可以当做 id 使用.

* 整合 struts1 的时候:

```
<bean name="/loginAction" >
```

1.2.5.2 scope 属性: Bean 的作用范围.

* singleton :默认值, 单例的.

* prototype :多例的.

* request :WEB 项目中, Spring 创建一个 Bean 的对象, 将对象存入到 request 域中.

* session :WEB 项目中, Spring 创建一个 Bean 的对象, 将对象存入到 session 域中.

* globalSession :WEB 项目中, [应用在 Porlet 环境](#).如果没有 [Porlet 环境那么](#) globalSession 相当于 session.

1.2.5.3 Bean 的生命周期的配置:

通过配置<bean>标签上的 init-method 作为 Bean 的初始化的时候执行的方法, 配置 destroy-method 作为 Bean 的销毁的时候执行的方法.

销毁方法想要执行, 需要是单例创建的 Bean 而且在工厂关闭的时候, Bean 才会被销毁.

1.2.6 Spring 的 Bean 的管理 XML 的方式:

1.2.6.1 Spring 生成 Bean 的时候三种方式(了解)

【无参数的构造方法的方式:】

<!-- 方式一: 无参数的构造方法的实例化 -->

```
<bean id="bean1" class="cn.itcast.spring.demo3.Bean1"></bean>
```

【静态工厂实例化的方式】

提供一个工厂类:

```
public class Bean2Factory {
```

```
        public static Bean2 getBean2() {  
            return new Bean2();  
        }  
    }  
  
<!-- 方式二：静态工厂实例化 Bean -->  
<bean id="bean2" class="cn.itcast.spring.demo3.Bean2Factory"  
factory-method="getBean2"/>
```

【实例工厂实例化的方式】

提供 Bean3 的实例工厂：

```
public class Bean3Factory {  
  
    public Bean3 getBean3() {  
        return new Bean3();  
    }  
}  
  
<!-- 方式三：实例工厂实例化 Bean -->  
<bean id="bean3Factory" class="cn.itcast.spring.demo3.Bean3Factory"></bean>  
<bean id="bean3" factory-bean="bean3Factory" factory-method="getBean3"></bean>
```

1.2.6.2 Spring 的 Bean 的属性注入：

【构造方法的方式注入属性】

```
<!-- 第一种：构造方法的方式 -->  
<bean id="car" class="cn.itcast.spring.demo4.Car">  
    <constructor-arg name="name" value="保时捷"/>  
    <constructor-arg name="price" value="1000000"/>  
</bean>
```

【set 方法的方式注入属性】

```
<!-- 第二种：set 方法的方式 -->  
<bean id="car2" class="cn.itcast.spring.demo4.Car2">  
    <property name="name" value="奇瑞 QQ"/>  
    <property name="price" value="40000"/>  
</bean>
```

1.2.6.3 Spring 的属性注入：对象类型的注入：

```
<!-- 注入对象类型的属性 -->  
<bean id="person" class="cn.itcast.spring.demo4.Person">
```

```
<property name="name" value="会希"/>
<!-- ref 属性: 引用另一个 bean 的 id 或 name -->
<property name="car2" ref="car2"/>
</bean>
```

1.2.6.4 名称空间 p 的属性注入的方式:Spring2.x 版本后提供的方式.

第一步:引入 p 名称空间

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

第二步:使用 p 名称空间.

- * 普通属性: p:属性名称=""
- * 对象类型属性: p:属性名称-ref=""

```
<!-- p 名称空间的属性注入的方式 -->
<bean id="car2" class="cn.itcast.spring.demo4.Car2" p:name=" 宝 马 7"
p:price="1200000"/>
<bean id="person" class="cn.itcast.spring.demo4.Person" p:name=" 思 聪 "
p:car2-ref="car2"/>
```

1.2.6.5 SpEL 的方式的属性注入:Spring3.x 版本后提供的方式.

SpEL: Spring Expression Language.

语法:#{ SpEL }

```
<!-- SpEL 的注入的方式 -->
<bean id="car2" class="cn.itcast.spring.demo4.Car2">
    <property name="name" value="#{'奔驰'}"/>
    <property name="price" value="#{800000}"/>
</bean>

<bean id="person" class="cn.itcast.spring.demo4.Person">
    <property name="name" value="#{'冠希'}"/>
    <property name="car2" value="#{car2}"/>
</bean>

<bean id="carInfo" class="cn.itcast.spring.demo4.CarInfo"></bean>
```

引用了另一个类的属性

```
<bean id="car2" class="cn.itcast.spring.demo4.Car2">
<!--      <property name="name" value="#{'奔驰'}"/> -->
      <property name="name" value="#{carInfo.carName}"/>
      <property name="price" value="#{carInfo.calculatePrice()}"/>
</bean>
```

1.2.6.6 注入复杂类型:

```
<!-- Spring 的复杂类型的注入===== -->
<bean id="collectionBean" class="cn.itcast.spring.demo5.CollectionBean">
  <!-- 数组类型的属性 -->
  <property name="arrs">
    <list>
      <value>会希</value>
      <value>冠希</value>
      <value>天一</value>
    </list>
  </property>

  <!-- 注入 List 集合的数据 -->
  <property name="list">
    <list>
      <value>芙蓉</value>
      <value>如花</value>
      <value>凤姐</value>
    </list>
  </property>

  <!-- 注入 Map 集合 -->
  <property name="map">
    <map>
      <entry key="aaa" value="111"/>
      <entry key="bbb" value="222"/>
      <entry key="ccc" value="333"/>
    </map>
  </property>

  <!-- Properties 的注入 -->
  <property name="properties">
    <props>
      <prop key="username">root</prop>
      <prop key="password">123</prop>
```

```
        </props>
    </property>
</bean>
```

1.2.6.7 Spring 的分配文件的开发

一种:创建工厂的时候加载多个配置文件:

```
ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml","applicationContext2.xml");
```

二种:在一个配置文件中包含另一个配置文件:

```
<import resource="applicationContext2.xml"></import>
```

1.3 案例代码

1.3.1 搭建环境:

1.3.1.1 创建 web 项目, 引入 jar 包.

WEB 层使用 Struts2:

- * Struts2 开发的基本的包

Spring 进行 Bean 管理:

- * Spring 开发的基本的包

1.3.1.2 引入配置文件:

```
Struts2:
    * web.xml

    <filter>
        <filter-name>struts2</filter-name>

        <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

```
* struts.xml

Spring:
* applicationContext.xml
* log4j.properties
```

1.3.1.3 引入页面:

1.3.1.4 创建包结构和类:



1.3.1.5 在添加页面提交内容到 Action:

```
<FORM id=form1 name=form1
  action="${pageContext.request.contextPath }/customer_save.action"
  method=post>
```

1.3.1.6 改写 Action 类并配置 Action:

```
<struts>
  <package name="crm" extends="struts-default" namespace="/">
    <action name="customer_*" class="cn.itcast.crm.web.action.CustomerAction"
method="{1}">
```

```
</action>
</package>
</struts>
```

1.3.1.7 在 Action 调用业务层:

将业务层类配置到 Spring 中:

```
<bean id="customerService"
class="cn.itcast.crm.service.impl.CustomerServiceImpl">

</bean>
```

在 Action 中获取业务层类:

```
public String save() {
    System.out.println("Action 中的 save 方法执行了...");
    System.out.println(customer);

    // 传统方式:
    /*CustomerService customerService = new CustomerServiceImpl();
    customerService.save(customer);*/

    // Spring 的方式进行操作:
    ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("applicationContext.xml");
    CustomerService customerService = (CustomerService)
applicationContext.getBean("customerService");
    customerService.save(customer);
    return NONE;
}
```

**** 每次请求都会创建一个工厂类,服务器端的资源就浪费了,一般情况下一个工程只有一个 Spring 的工厂类就 OK 了。

* 将工厂在服务器启动的时候创建好,将这个工厂放入到 ServletContext 域中.每次获取工厂从 ServletContext 域中进行获取。

* ServletContextListener :监听 ServletContext 对象的创建和销毁。

1.3.2 Spring 整合 WEB 项目

1.3.2.1 引入 spring-web.jar 包:

配置监听器:

```
<listener>
```



```
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-  
class>  
    </listener>  
  
    <context-param>  
        <param-name>contextConfigLocation</param-name>  
        <param-value>classpath:applicationContext.xml</param-value>  
    </context-param>
```

1.3.2.2 改写 Action:

```
/**  
 * 保存客户的执行的方法: save  
 */  
public String save() {  
    // 传统方式:  
    /*CustomerService customerService = new CustomerServiceImpl();  
    customerService.save(customer);*/  
  
    // Spring 的方式进行操作:  
    /*ApplicationContext applicationContext = new  
ClassPathXmlApplicationContext("applicationContext.xml");  
    CustomerService customerService = (CustomerService)  
applicationContext.getBean("customerService");*/  
  
    WebApplicationContext applicationContext  
=WebApplicationContextUtils.getWebApplicationContext(ServletActionContext.getServlet  
Context());  
    CustomerService customerService = (CustomerService)  
applicationContext.getBean("customerService");  
  
    System.out.println("Action 中的 save 方法执行了...");  
    System.out.println(customer);  
    customerService.save(customer);  
    return NONE;  
}
```

1.3.2.3 编写 Dao 并配置:

```
<bean id="customerDao" class="cn.itcast.crm.dao.impl.CustomerDaoImpl">

</bean>
```

1.3.2.4 业务层调用 DAO:

```
public class CustomerServiceImpl implements CustomerService {

    private CustomerDao customerDao;

    public void setCustomerDao(CustomerDao customerDao) {
        this.customerDao = customerDao;
    }

    ...
}

<bean id="customerService"
class="cn.itcast.crm.service.impl.CustomerServiceImpl">
    <property name="customerDao" ref="customerDao"/>
</bean>
```

Spring_day02 总结

今日任务

- 使用 Spring 的 AOP 对客户管理的 DAO 进行增强

教学导航

教学目标	
教学方法	案例驱动法

案例一使用 Spring 的 AOP 对客户管理的 DAO 进行增强

1.1 案例需求

1.1.1 需求描述

对于 CRM 的系统而言，现在有很多的 DAO 类，比如客户的 DAO，联系人 DAO 等等。客户提出一个需求要开发人员实现一个功能对所有的 DAO 的类中以 save 开头的方法实现权限的校验，需要时管理员的身份才可以进行保存操作。

1.2 相关知识点

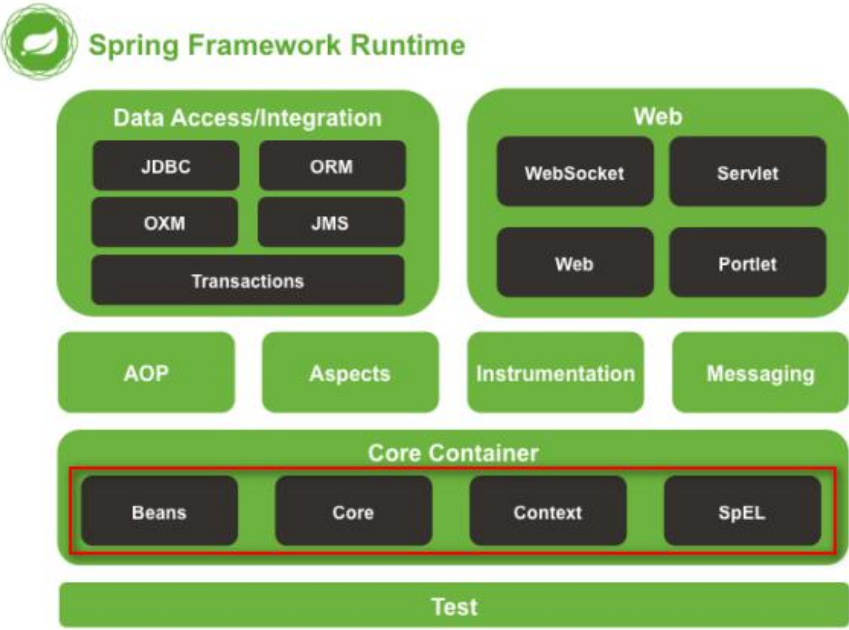
1.2.1 Spring 的 Bean 管理:(注解方式)

1.2.1.1 步骤一:下载 Spring 的开发包:

官网: <http://spring.io/>
下载地址: <http://repo.springsource.org/libs-release-local/org/springframework/spring> 解压: (Spring 目录结构:)
* docs :API 和开发规范.
* libs :jar 包和源码.

* schema :约束.

1.2.1.2 步骤二:创建 web 项目,引入 Spring 的开发包:



The diagram illustrates the Spring Framework Runtime architecture. It is organized into several layers:

- Data Access/Integration**: Includes JDBC, ORM, OXM, JMS, and Transactions.
- Web**: Includes WebSocket, Servlet, Web, and Portlet.
- Core Container**: Includes Beans, Core, Context, and SpEL. This layer is highlighted with a red border.
- Test**: The base layer.
- Other components**: AOP, Aspects, Instrumentation, and Messaging.

Below the diagram, a list of JAR files is shown in a 'lib' directory:

- com.springsource.org.apache.commons.logging-1.1.1.jar
- com.springsource.org.apache.log4j-1.2.15.jar
- spring-beans-4.2.4.RELEASE.jar
- spring-context-4.2.4.RELEASE.jar
- spring-core-4.2.4.RELEASE.jar
- spring-expression-4.2.4.RELEASE.jar

在 Spring 的注解的 AOP 中需要引入 spring-aop 的 jar 包。

1.2.1.3 步骤三:引入相关配置文件:

```
log4j.properties
applicationContext.xml
引入约束:
spring-framework-4.2.4.RELEASE\docs\spring-framework-reference\html\xsd-configuration.html

* 引入约束: (引入 context 的约束):
<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"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
```

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

</beans>
```

1.2.1.4 步骤四:编写相关的类:

```
public interface UserDao {

    public void sayHello();
}

public class UserDaoImpl implements UserDao {

    @Override
    public void sayHello() {
        System.out.println("Hello Spring...");
    }

}
```

1.2.1.5 步骤五:配置注解扫描

```
<!-- Spring 的注解开发:组件扫描 (类上注解: 可以直接使用属性注入的注解) -->
<context:component-scan base-package="com.itheima.spring.demo1"/>
```

1.2.1.6 在相关的类上添加注解:

```
@Component(value="userDao")
public class UserDaoImpl implements UserDao {

    @Override
    public void sayHello() {
        System.out.println("Hello Spring Annotation...");
    }

}
```

1.2.1.7 编写测试类:

```
@Test
public void demo2() {
    ApplicationContext applicationContext = new
    ClassPathXmlApplicationContext(
        "applicationContext.xml");
    UserDao userDao = (UserDao) applicationContext.getBean("userDao");
    userDao.sayHello();
}
```

1.2.2 Spring 的 Bean 管理中常用的注解:

1.2.2.1 @Component:组件.(作用在类上)

Spring 中提供@Component 的三个衍生注解: (功能目前来讲是一致的)

- * @Controller :WEB 层
- * @Service :业务层
- * @Repository :持久层

这三个注解是为了让标注类本身的用途清晰, Spring 在后续版本会对其增强

1.2.2.2 属性注入的注解:(使用注解注入的方式,可以不用提供 set 方法.)

@Value :用于注入普通类型.
@Autowired :自动装配:
* 默认按类型进行装配.
* 按名称注入:
* @Qualifier:强制使用名称注入.
@Resource 相当于:
* @Autowired 和@Qualifier 一起使用.

1.2.2.3 Bean 的作用范围的注解:

@Scope:
* singleton:单例
* prototype:多例

1.2.2.4 Bean 的生命周期的配置:

@PostConstruct :相当于 init-method
@PreDestroy :相当于 destroy-method

1.2.3 Spring 的 Bean 管理的方式的比较:

	基于XML配置	基于注解配置
Bean定义	<bean id="..." class="..." />	@Component 衍生类@Repository @Service @Controller
Bean名称	通过 id或name 指定	@Component("person")
Bean注入	<property> 或者 通过p命名空间	@Autowired 按类型注入 @Qualifier按名称注入
生命过程、 Bean作用范围	init-method destroy-method 范围 scope属性	@PostConstruct 初始化 @PreDestroy 销毁 @Scope设置作用范围
适合场景	Bean来自第三 方, 使用其它	Bean的实现类由用户自己 开发

XML 和注解:

- * XML :结构清晰.
- * 注解 :开发方便.(属性注入.)

实际开发中还有一种 XML 和注解整合开发:

- * Bean 有 XML 配置,但是使用的属性使用注解注入.

1.2.4 AOP 的概述

1.2.4.1 什么是 AOP

AOP (面向切面编程)

编辑

在软件业，AOP为Aspect Oriented Programming的缩写，意为：面向切面编程，通过预编译方式和运行期功能的统一维护的一种技术。AOP是OOP的延续，是软件开发中的一个热点，也是Spring框架中的一个重要概念。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高重用性，同时提高了开发的效率。

Spring 是解决实际开发中的一些问题：

- * AOP 解决 OOP 中遇到的一些问题，是 OOP 的延续和扩展。

1.2.4.2 为什么学习 AOP

对程序进行增强：不修改源码的情况下。

- * AOP 可以进行权限校验，日志记录，性能监控，事务控制。

1.2.4.3 Spring 的 AOP 的由来：

AOP 最早由 AOP 联盟的组织提出的，制定了一套规范。Spring 将 AOP 思想引入到框架中，必须遵守 AOP 联盟的规范。

1.2.4.4 底层实现：

代理机制：

- * Spring 的 AOP 的底层用到两种代理机制：

- * JDK 的动态代理：针对实现了接口的类产生代理。
- * Cglib 的动态代理：针对没有实现接口的类产生代理。应用的是底层的字节码增强的技术 生成当前类的子类对象。

1.2.5 Spring 底层 AOP 的实现原理：（了解）

1.2.5.1 JDK 动态代理增强一个类中方法：

```
public class MyJDKProxy implements InvocationHandler {
```

```
private UserDao userDao;

public MyJDKProxy(UserDao userDao) {
    this.userDao = userDao;
}

// 编写工具方法：生成代理：
public UserDao createProxy() {
    UserDao userDaoProxy = Proxy.newProxyInstance(userDao.getClass().getClassLoader(),
        userDao.getClass().getInterfaces(), this);

    return userDaoProxy;
}

@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    if ("save".equals(method.getName())) {
        System.out.println("权限校验=====");
    }

    return method.invoke(userDao, args);
}
}
```

1.2.5.2 Cglib 动态代理增强一个类中的方法:

```
public class MyCglibProxy implements MethodInterceptor{

    private CustomerDao customerDao;

    public MyCglibProxy(CustomerDao customerDao){
        this.customerDao = customerDao;
    }

    // 生成代理的方法：
    public CustomerDao createProxy(){
        // 创建 Cglib 的核心类：
        Enhancer enhancer = new Enhancer();
        // 设置父类：
        enhancer.setSuperclass(CustomerDao.class);
        // 设置回调：
        enhancer.setCallback(this);
        // 生成代理：
    }
}
```

```
        CustomerDao customerDaoProxy = (CustomerDao) enhancer.create();
        return customerDaoProxy;
    }

    @Override
    public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {
        if ("delete".equals(method.getName())) {
            Object obj = methodProxy.invokeSuper(proxy, args);
            System.out.println("日志记录=====");
            return obj;
        }

        return methodProxy.invokeSuper(proxy, args);
    }
}
```

1.2.6 Spring 的基于 AspectJ 的 AOP 开发

1.2.6.1 AOP 的开发中的相关术语:

Joinpoint (连接点): 所谓连接点是指那些被拦截到的点。在 spring 中, 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): 所谓切入点是指我们要对哪些 Joinpoint 进行拦截的定义。

Advice (通知/增强): 所谓通知是指拦截到 Joinpoint 之后所要做的事情就是通知。通知分为前置通知, 后置通知, 异常通知, 最终通知, 环绕通知 (切面要完成的功能)

Introduction (引介): 引介是一种特殊的通知在不修改类代码的前提下, Introduction 可以在运行期为类动态地添加一些方法或 Field。

Target (目标对象): 代理的目标对象

Weaving (织入): 是指把增强应用到目标对象来创建新的代理对象的过程。

spring 采用动态代理织入, 而 AspectJ 采用编译期织入和类装载期织入

Proxy (代理): 一个类被 AOP 织入增强后, 就产生一个结果代理类

Aspect (切面): 是切入点和通知 (引介) 的结合

1.2.7 Spring 使用 AspectJ 进行 AOP 的开发: XML 的方式 (*****)

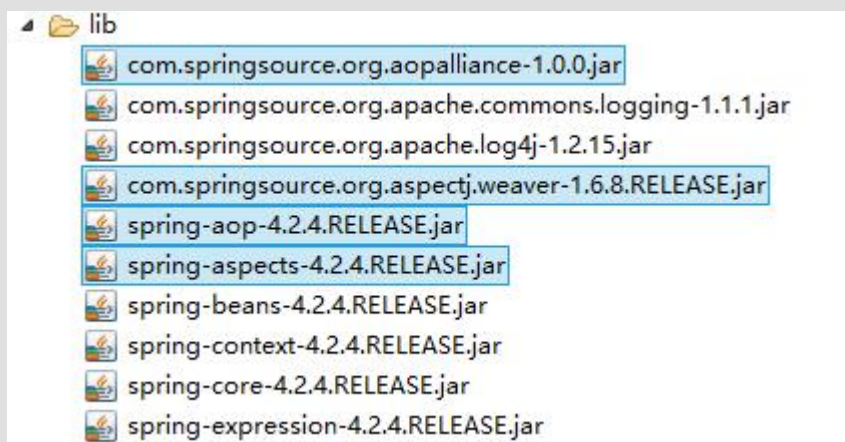
1.2.7.1 引入相应的 jar 包

```
* spring 的传统 AOP 的开发的包
spring-aop-4.2.4.RELEASE.jar
com.springsource.org.aopalliance-1.0.0.jar
```

* aspectJ 的开发包：

com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

spring-aspects-4.2.4.RELEASE.jar



1.2.7.2 引入 Spring 的配置文件

引入 AOP 约束：

```
<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" xsi:schemaLocation="
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">

  </beans>
```

1.2.7.3 编写目标类

创建接口和类：

```
public interface OrderDao {
    public void save();
    public void update();
    public void delete();
    public void find();
}

public class OrderDaoImpl implements OrderDao {

    @Override
    public void save() {
        System.out.println("保存订单...");
    }
}
```

```
@Override
public void update() {
    System.out.println("修改订单...");
}

@Override
public void delete() {
    System.out.println("删除订单...");
}

@Override
public void find() {
    System.out.println("查询订单...");
}
}
```

1.2.7.4 目标类的配置

```
<!-- 目标类===== -->
<bean id="orderDao" class="cn.itcast.spring.demo3.OrderDaoImpl">

</bean>
```

1.2.7.5 整合 Junit 单元测试

引入 spring-test.jar

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class SpringDemo3 {
    @Resource(name="orderDao")
    private OrderDao orderDao;

    @Test
    public void demo1(){
        orderDao.save();
        orderDao.update();
        orderDao.delete();
        orderDao.find();
    }
}
```

1.2.7.6 通知类型

前置通知：在目标方法执行之前执行。
后置通知：在目标方法执行之后执行
环绕通知：在目标方法执行前和执行后执行
异常抛出通知：在目标方法执行出现异常的时候执行
最终通知：无论目标方法是否出现异常，最终通知都会执行。

1.2.7.7 切入点表达式

execution(表达式)
表达式：
[方法访问修饰符] 方法返回值 包名.类名.方法名 (方法的参数)
public * cn.itcast.spring.dao.*.*(..)
* cn.itcast.spring.dao.*.*(..)
* cn.itcast.spring.dao.UserDao+.*(..)
* cn.itcast.spring.dao..*.*(..)

1.2.7.8 编写一个切面类

```
public class MyAspectXml {  
    // 前置增强  
    public void before() {  
        System.out.println("前置增强=====");  
    }  
}
```

1.2.7.9 配置完成增强

```
<!-- 配置切面类 -->  
<bean id="myAspectXml" class="cn.itcast.spring.demo3.MyAspectXml"></bean>  
  
<!-- 进行 aop 的配置 -->  
<aop:config>  
    <!-- 配置切入点表达式:哪些类的哪些方法需要进行增强 -->  
    <aop:pointcut expression="execution(*  
cn.itcast.spring.demo3.OrderDao.save(..)" id="pointcut1"/>  
    <!-- 配置切面 -->  
    <aop:aspect ref="myAspectXml">  
        <aop:before method="before" pointcut-ref="pointcut1"/>  
    </aop:aspect>  
</aop:config>
```

1.2.7.10 其他的增强的配置：

```
<!-- 配置切面类 -->
<bean id="myAspectXml" class="cn.itcast.spring.demo3.MyAspectXml"></bean>

<!-- 进行 aop 的配置 -->
<aop:config>
    <!-- 配置切入点表达式:哪些类的哪些方法需要进行增强 -->
    <aop:pointcut expression="execution(*
cn.itcast.spring.demo3.*Dao.save(..))" id="pointcut1"/>
    <aop:pointcut expression="execution(*
cn.itcast.spring.demo3.*Dao.delete(..))" id="pointcut2"/>
    <aop:pointcut expression="execution(*
cn.itcast.spring.demo3.*Dao.update(..))" id="pointcut3"/>
    <aop:pointcut expression="execution(*
cn.itcast.spring.demo3.*Dao.find(..))" id="pointcut4"/>
    <!-- 配置切面 -->
    <aop:aspect ref="myAspectXml">
        <aop:before method="before" pointcut-ref="pointcut1"/>
        <aop:after-returning method="afterReturing"
pointcut-ref="pointcut2"/>
        <aop:around method="around" pointcut-ref="pointcut3"/>
        <aop:after-throwing method="afterThrowing" pointcut-ref="pointcut4"/>
        <aop:after method="after" pointcut-ref="pointcut4"/>
    </aop:aspect>
</aop:config>
```


Spring_day03 总结

今日任务

- 使用 Spring 的 AOP 对客户管理的 DAO 进行增强
- 使用 Spring 完成转账的事务管理

教学导航

教学目标	掌握 Spring 的声明式事务 掌握 SSH 的整合
教学方法	案例驱动法

案例一：使用 Spring 的 AOP 对客户管理的 DAO 进行增强

1.1 案例需求

1.1.1 需求描述

对于 CRM 的系统而言，现在有很多的 DAO 类，比如客户的 DAO，联系人 DAO 等等。客户提出一个需求要开发人员实现一个功能对所有的 DAO 的类中以 save 开头的方法实现权限的校验，需要时管理员的身份才可以进行保存操作。

1.2 相关知识点

1.2.1 Spring 使用 AspectJ 进行 AOP 的开发:注解的方式

1.2.1.1 引入相关的 jar 包:

```
* spring 的传统 AOP 的开发的包
spring-aop-4.2.4.RELEASE.jar
com.springsource.org.aopalliance-1.0.0.jar
* aspectJ 的开发包:
```

```
com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar
spring-aspects-4.2.4.RELEASE.jar
```



1.2.1.2 引入 Spring 的配置文件

引入 AOP 约束：

```
<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" xsi:schemaLocation="
            http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">

    </beans>
```

1.2.1.3 编写目标类：

```
public class ProductDao {
    public void save() {
        System.out.println("保存商品...");
    }
    public void update() {
        System.out.println("修改商品...");
    }
    public void delete() {
        System.out.println("删除商品...");
    }
    public void find() {
        System.out.println("查询商品...");
    }
}
```

1.2.1.4 配置目标类:

```
<!-- 目标类===== -->
<bean id="productDao" class="cn.itcast.spring.demo4.ProductDao"></bean>
```

1.2.1.5 开启 aop 注解的自动代理:

```
<aop:aspectj-autoproxy/>
```

1.2.1.6 AspectJ 的 AOP 的注解:

@Aspect: 定义切面类的注解

通知类型:

* @Before	: 前置通知
* @AfterReturing	: 后置通知
* @Around	: 环绕通知
* @After	: 最终通知
* @AfterThrowing	: 异常抛出通知.

@Pointcut: 定义切入点的注解

1.2.1.7 编写切面类:

```
@Aspect
public class MyAspectAnno {

    @Before("MyAspectAnno.pointcut1()")
    public void before() {
        System.out.println("前置通知=====");
    }

    @Pointcut("execution(* cn.itcast.spring.demo4.ProductDao.save(..))")
    private void pointcut1() {}
}
```

1.2.1.8 配置切面:

```
<!-- 配置切面类 -->
<bean id="myAspectAnno" class="cn.itcast.spring.demo4.MyAspectAnno"></bean>
```

1.2.1.9 其他通知的注解:

@Aspect

```
public class MyAspectAnno {

    @Before("MyAspectAnno.pointcut1()")
    public void before() {
        System.out.println("前置通知=====");
    }

    @AfterReturning("MyAspectAnno.pointcut2()")
    public void afterReturning() {
        System.out.println("后置通知=====");
    }

    @Around("MyAspectAnno.pointcut3()")
    public Object around(ProceedingJoinPoint joinPoint) throws Throwable{
        System.out.println("环绕前通知=====");
        Object obj = joinPoint.proceed();
        System.out.println("环绕后通知=====");
        return obj;
    }

    @AfterThrowing("MyAspectAnno.pointcut4()")
    public void afterThrowing() {
        System.out.println("异常抛出通知=====");
    }

    @After("MyAspectAnno.pointcut4()")
    public void after() {
        System.out.println("最终通知=====");
    }

    @Pointcut("execution(* cn.itcast.spring.demo4.ProductDao.save(..))")
    private void pointcut1() {}
    @Pointcut("execution(* cn.itcast.spring.demo4.ProductDao.update(..))")
    private void pointcut2() {}
    @Pointcut("execution(* cn.itcast.spring.demo4.ProductDao.delete(..))")
    private void pointcut3() {}
    @Pointcut("execution(* cn.itcast.spring.demo4.ProductDao.find(..))")
    private void pointcut4() {}
}
```

案例二：Spring 的事务管理完成转账的案例

1.3 案例需求:

1.3.1 需求描述:

完成一个转账的功能,需要进行事务的管理,使用 Spring 的事务管理的方式完成.

1.4 相关知识点

1.4.1 Spring 的 JDBC 的模板:

1.4.1.1 Spring 提供了很多持久层技术的模板类简化编程:

ORM持久化技术	模板类
JDBC	org.springframework.jdbc.core.JdbcTemplate
Hibernate3.0	org.springframework.orm.hibernate3.HibernateTemplate
IBatis(MyBatis)	org.springframework.orm.ibatis.SqlMapClientTemplate
JPA	org.springframework.orm.jpa.JpaTemplate

1.4.1.2 创建数据库和表:

1.4.1.3 引入相关开发包:

Spring 的基本的开发包需要引入的:6 个.



1.4.1.4 创建一个测试类:

```
@Test
// JDBC 模板的基本使用:
public void demo1() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql:///spring_day03");
    dataSource.setUsername("root");
    dataSource.setPassword("123");

    JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
    jdbcTemplate.update("insert into account values (null,?,?)", "会希", 10000d);
}
```

1.4.2 将连接池的配置交给 Spring 管理:

1.4.2.1 Spring 内置的连接池的配置:

【引入 Spring 的配置文件】

【配置内置连接池】

```
<!-- 配置 Spring 的内置连接池 -->
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql:///spring_day02"/>
    <property name="username" value="root"/>
    <property name="password" value="123"/>
</bean>
```

【将模板配置到 Spring 中】

```
<!-- 配置 JDBC 模板 -->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

【编写测试类】

```
**** 引入 spring-aop.jar

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class SpringDemo2 {

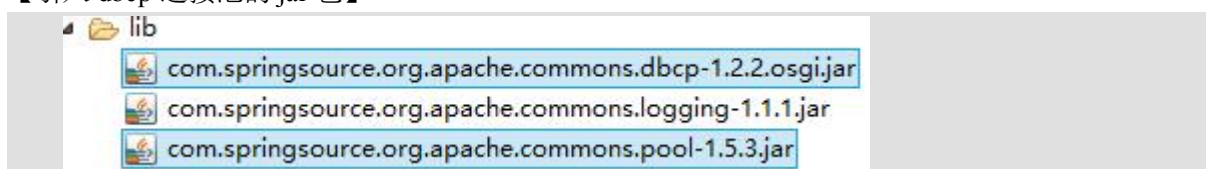
    @Resource(name="jdbcTemplate")
    private JdbcTemplate jdbcTemplate;

    @Test
    public void demo1() {
        jdbcTemplate.update("insert into account values (null,?,?)", "凤姐", 10000d);
    }

}
```

1.4.2.2 Spring 中配置 DBCP 连接池:

【引入 dbcp 连接池的 jar 包】



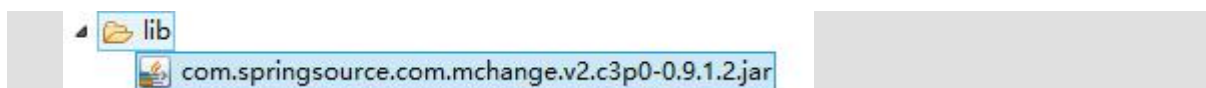
【配置连接池】

```
<!-- 配置 DBCP 连接池 -->
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql:///spring_day02"/>
    <property name="username" value="root"/>
    <property name="password" value="123"/>
</bean>
```

1.4.2.3 配置 c3p0 连接池:

【引入相应的 jar 包】

```
com.springsource.com.mchange.v2.c3p0-0.9.1.2.jar
```

【配置连接池】

```
<!-- 配置 C3P0 连接池 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
  <property name="driverClass" value="com.mysql.jdbc.Driver"/>
  <property name="jdbcUrl" value="jdbc:mysql:///spring_day02"/>
  <property name="user" value="root"/>
  <property name="password" value="123"/>
</bean>
```

1.4.2.4 将数据库连接的信息配置到属性文件中:

【定义属性文件】

```
jdbc.driverClass=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql:///spring_day02
jdbc.username=root
jdbc.password=123
```

【引入外部的属性文件】

一种方式:

```
<!-- 引入外部属性文件: -->
<bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="location" value="classpath:jdbc.properties"/>
</bean>
```

二种方式:

```
<context:property-placeholder location="classpath:jdbc.properties"/>
```

1.4.3 JDBC 模板的 CRUD 的操作:

1.4.3.1 JDBC 模板 CRUD 的操作:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class SpringDemo3 {

  @Resource(name="jdbcTemplate")
  private JdbcTemplate jdbcTemplate;

  @Test
  // 插入操作
  public void demo1() {
```

```
jdbcTemplate.update("insert into account values (null,?,?)", "冠希",10000d);
    }

    @Test
    // 修改操作
    public void demo2() {
        jdbcTemplate.update("update account set name=?,money =? where id = ?", "思雨",10000d,5);
    }

    @Test
    // 删除操作
    public void demo3() {
        jdbcTemplate.update("delete from account where id = ?", 5);
    }

    @Test
    // 查询一条记录
    public void demo4() {
        Account account = jdbcTemplate.queryForObject("select * from account where id = ?", new MyRowMapper(), 1);
        System.out.println(account);
    }

    @Test
    // 查询所有记录
    public void demo5() {
        List<Account> list = jdbcTemplate.query("select * from account", new MyRowMapper());

        for (Account account : list) {
            System.out.println(account);
        }
    }

    class MyRowMapper implements RowMapper<Account>{

        @Override
        public Account mapRow(ResultSet rs, int rowNum) throws SQLException {
            Account account = new Account();
            account.setId(rs.getInt("id"));
            account.setName(rs.getString("name"));
            account.setMoney(rs.getDouble("money"));
            return account;
        }
    }
}
```

```
    }  
  
    }  
  
}
```

1.4.4 事务的回顾:

1.4.4.1 什么是事务:

事务逻辑上的一组操作,组成这组操作的各个逻辑单元,要么一起成功,要么一起失败.

1.4.4.2 事务特性:

原子性 :强调事务的不可分割.

一致性 :事务的执行的前后数据的完整性保持一致.

隔离性 :一个事务执行的过程中,不应该受到其他事务的干扰

持久性 :事务一旦结束,数据就持久到数据库

1.4.4.3 如果不考虑隔离性引发安全性问题:

脏读 :一个事务读到了另一个事务的未提交的数据

不可重复读 :一个事务读到了另一个事务已经提交的 update 的数据导致多次查询结果不一致.

虚读 :一个事务读到了另一个事务已经提交的 insert 的数据导致多次查询结果不一致.

1.4.4.4 解决读问题:设置事务隔离级别

未提交读 :脏读, 不可重复读, 虚读都有可能发生

已提交读 :避免脏读.但是不可重复读和虚读有可能发生

可重复读 :避免脏读和不可重复读.但是虚读有可能发生.

串行化的 :避免以上所有读问题.

1.4.5 Spring 进行事务管理一组 API

1.4.5.1 PlatformTransactionManager:平台事务管理器.

```
***** 真正管理事务的对象  
org.springframework.jdbc.datasource.DataSourceTransactionManager 使用 Spring  
JDBC 或 iBatis 进行持久化数据时使用
```

`org.springframework.orm.hibernate3.HibernateTransactionManager` 使用
Hibernate 版本进行持久化数据时使用

1.4.5.2 TransactionDefinition:事务定义信息

事务定义信息:

* 隔离级别

* 传播行为

* 超时信息

* 是否只读

1.4.5.3 TransactionStatus:事务的状态

记录事务的状态

1.4.5.4 Spring 的这组接口是如何进行事务管理:

平台事务管理根据事务定义的信息进行事务的管理,事务管理的过程中产生一些状态,将这些状态记录到 TransactionStatus 里面

1.4.5.5 事务的传播行为

PROPAGION_XXX :事务的传播行为

* 保证同一个事务中

PROPAGATION_REQUIRED 支持当前事务, 如果不存在 就新建一个 (默认)

PROPAGATION_SUPPORTS 支持当前事务, 如果不存在, 就不使用事务

PROPAGATION_MANDATORY 支持当前事务, 如果不存在, 抛出异常

* 保证没有在同一个事务中

PROPAGATION_REQUIRES_NEW 如果有事务存在, 挂起当前事务, 创建一个新的事务

PROPAGATION_NOT_SUPPORTED 以非事务方式运行, 如果有事务存在, 挂起当前事务

PROPAGATION_NEVER 以非事务方式运行, 如果有事务存在, 抛出异常

PROPAGATION_NESTED 如果当前事务存在, 则嵌套事务执行

1.5 案例代码

1.5.1 搭建转账的环境:

1.5.1.1 创建业务层和 DAO 的类

```
public interface AccountService {

    public void transfer(String from,String to,Double money);

}

public class AccountServiceImpl implements AccountService {

    // 业务层注入 DAO:
    private AccountDao accountDao;

    public void setAccountDao(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    @Override
    /**
     * from:转出的账号
     * to:转入的账号
     * money: 转账金额
     */
    public void transfer(String from, String to, Double money) {
        accountDao.outMoney(from, money);
        accountDao.inMoney(to, money);
    }

}

public interface AccountDao {

    public void outMoney(String from,Double money);

    public void inMoney(String to,Double money);

}

public class AccountDaoImpl extends JdbcDaoSupport implements AccountDao {
```

```
@Override
    public void outMoney(String from, Double money) {
        this.getJdbcTemplate().update("update account set money = money - ? where
name = ?", money, from);
    }

@Override
    public void inMoney(String to, Double money) {
        this.getJdbcTemplate().update("update account set money = money + ? where
name = ?", money, to);
    }
}
```

1.5.1.2 配置业务层和 DAO

```
<!-- 配置业务层的类 -->
<bean id="accountService"
class="cn.itcast.transaction.demol.AccountServiceImpl">
    <property name="accountDao" ref="accountDao"/>
</bean>

<!-- 配置 DAO 的类 -->
<bean id="accountDao" class="cn.itcast.transaction.demol.AccountDaoImpl">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

1.5.1.3 编写测试类

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext2.xml")
public class SpringDemo4 {

    @Resource(name="accountService")
    private AccountService accountService;

    @Test
    // 转账的测试:
    public void demo1() {
        accountService.transfer("会希", "凤姐", 1000d);
    }
}
```

1.5.2 Spring 的编程式事务(了解)

手动编写代码完成事务的管理:

1.5.2.1 配置事务管理器

```
<!-- 配置事务管理器 -->
<bean                                id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

1.5.2.2 配置事务管理的模板

```
<!-- 配置事务管理模板 -->
<bean                                id="transactionTemplate"
class="org.springframework.transaction.support.TransactionTemplate">
    <property name="transactionManager" ref="transactionManager"/>
</bean>
```

1.5.2.3 需要在业务层注入事务管理模板

```
<!-- 配置业务层的类 -->
<bean                                id="accountService"
class="cn.itcast.transaction.demo1.AccountServiceImpl">
    <property name="accountDao" ref="accountDao"/>
    <!-- 注入事务管理模板 -->
    <property name="transactionTemplate" ref="transactionTemplate"/>
</bean>
```

1.5.2.4 手动编写代码实现事务管理

```
public void transfer(final String from, final String to, final Double money) {

    transactionTemplate.execute(new TransactionCallbackWithoutResult() {

        @Override
        protected void doInTransactionWithoutResult(TransactionStatus status)
        {

            accountDao.outMoney(from, money);
```

```
        int d = 1 / 0;
        accountDao.inMoney(to, money);

    }

});

}
```

1.5.3 Spring 的声明式事务管理 XML 方式(****): 思想就是 AOP.

不需要进行手动编写代码, 通过一段配置完成事务管理

1.5.3.1 引入 AOP 开发的包

```
aop 联盟.jar
Spring-aop.jar
aspectJ.jar
spring-aspects.jar
```

1.5.3.2 恢复转账环境

1.5.3.3 配置事务管理器

```
<!-- 事务管理器 -->
<bean                                id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

1.5.3.4 配置事务的通知

```
<!-- 配置事务的增强 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <!--
            isolation="DEFAULT"    隔离级别
            propagation="REQUIRED" 传播行为
        -->
```



```
        read-only="false"    只读
        timeout="-1"        过期时间
        rollback-for=""      -Exception
        no-rollback-for=""   +Exception
    -->

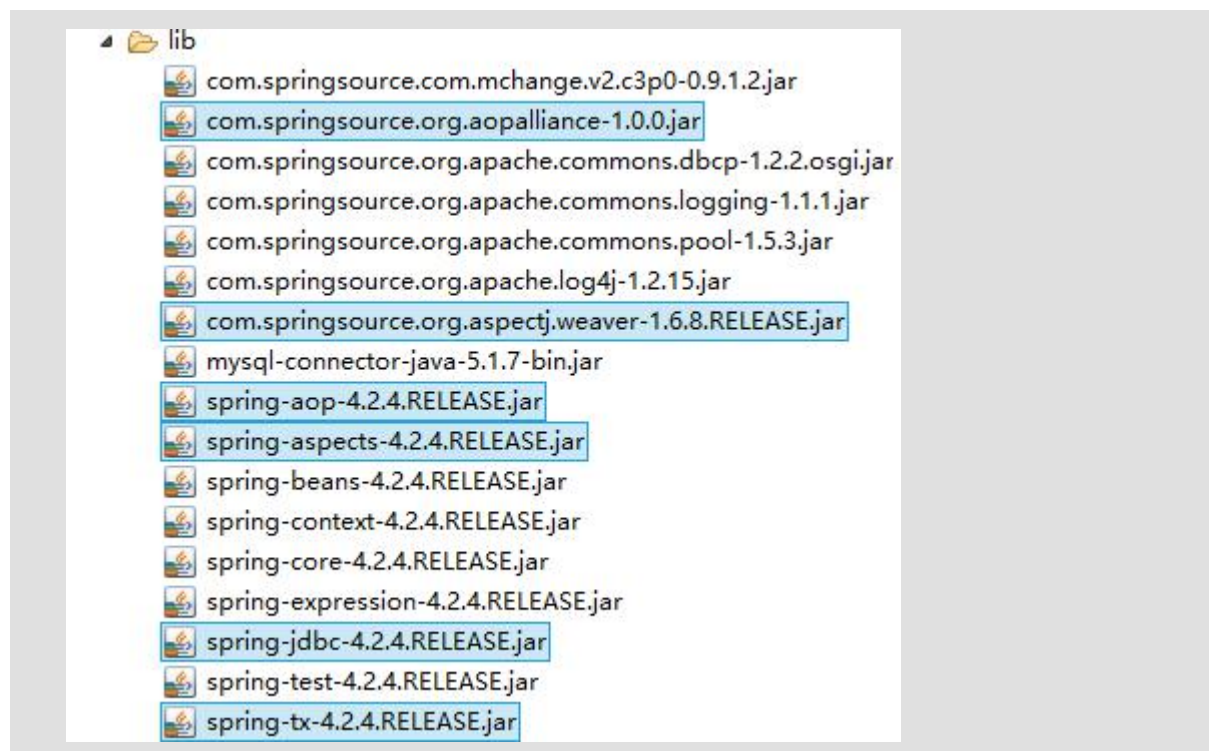
    <tx:method name="transfer" propagation="REQUIRED"/>
</tx:attributes>
</tx:advice>
```

1.5.3.5 配置 aop 事务

```
<aop:config>
    <aop:pointcut                                expression="execution(*
cn.itcast.transaction.demo2.AccountServiceImpl.transfer(..))" id="pointcut1"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="pointcut1"/>
</aop:config>
```

1.5.4 Spring 的声明式事务的注解方式: (****)

1.5.4.1 引入 jar 包:



1.5.4.2 恢复转账环境:

1.5.4.3 配置事务管理器:

```
<!-- 配置事务管理器 -->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

1.5.4.4 开启事务管理的注解:

```
<!-- 开启注解事务管理 -->
<tx:annotation-driven transaction-manager="transactionManager"/>
```

1.5.4.5 在使用事务的类上添加一个注解: @Transactional

```
@Transactional
public class AccountServiceImpl implements AccountService {
```

Spring_day04 总结

今日任务

- 使用 SSH 整合完成客户的保存操作

教学导航

教学目标	
教学方法	案例驱动法

案例一使用 SSH 的整合完成客户的保存操作

1.1 案例需求

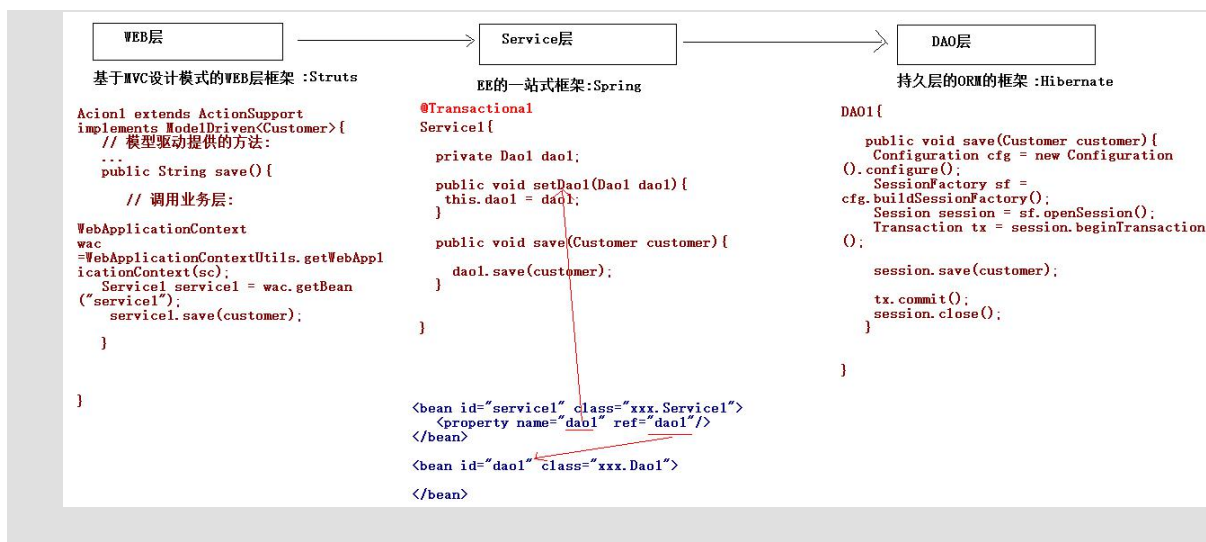
1.1.1 需求描述

使用 SSH 整合完成 CRM 的客户保存操作

1.2 相关知识点:

1.2.1 SSH 简单的回顾:

1.2.1.1 SSH 的基本开发回顾



1.2.2 SSH 框架的整合方式一: 零障碍整合(带有 Hibernate 配置文件)

1.2.2.1 创建 web 项目, 引入相关 jar 包.

【Struts2】

D:\struts2\struts-2.3.24\apps\struts2-blank\WEB-INF\lib*.jar










asm-3.3.jar	2013/11/23 17:55	Executable Jar File	43 KB
asm-commons-3.3.jar	2013/11/23 17:55	Executable Jar File	38 KB
asm-tree-3.3.jar	2013/11/23 17:55	Executable Jar File	21 KB
commons-fileupload-1.3.1.jar	2014/2/19 16:21	Executable Jar File	68 KB
commons-io-2.2.jar	2013/11/23 17:55	Executable Jar File	170 KB
commons-lang3-3.2.jar	2014/1/2 21:45	Executable Jar File	376 KB
freemarker-2.3.22.jar	2015/4/3 7:09	Executable Jar File	1,271 KB
javassist-3.11.0.GA.jar	2013/11/23 17:55	Executable Jar File	600 KB
log4j-api-2.2.jar	2015/4/19 12:04	Executable Jar File	131 KB
log4j-core-2.2.jar	2015/4/19 12:04	Executable Jar File	808 KB
ognl-3.0.6.jar	2013/11/23 17:55	Executable Jar File	223 KB
struts2-core-2.3.24.jar	2015/5/3 12:25	Executable Jar File	813 KB
xwork-core-2.3.24.jar	2015/5/3 12:23	Executable Jar File	661 KB

Struts2 需要了解的 jar 包:



struts2-convention-plugin-2.3.24.jar	---Struts2 注解的开发包.
struts2-json-plugin-2.3.24.jar	---Struts2 整合 AJAX 返回 JSON 数据.
struts2-spring-plugin-2.3.24.jar	---Struts2 整合 Spring 的插件包.

【Hibernate】

D:\hibernate-release-5.0.7.Final\lib\required*.jar

 antlr-2.7.7.jar	2014/4/28 20:30	Executable Jar File	435 KB
 dom4j-1.6.1.jar	2014/4/28 20:28	Executable Jar File	307 KB
 geronimo-jta_1.1_spec-1.1.1.jar	2015/5/5 11:26	Executable Jar File	16 KB
 hibernate-commons-annotations-5.0....	2015/11/30 10:22	Executable Jar File	74 KB
 hibernate-core-5.0.7.Final.jar	2016/1/13 12:35	Executable Jar File	5,453 KB
 hibernate-jpa-2.1-api-1.0.0.Final.jar	2014/4/28 20:30	Executable Jar File	111 KB
 jandex-2.0.0.Final.jar	2015/11/30 10:22	Executable Jar File	184 KB
 javassist-3.18.1-GA.jar	2014/4/28 20:28	Executable Jar File	698 KB
 jboss-logging-3.3.0.Final.jar	2015/5/28 12:35	Executable Jar File	66 KB

日志记录:

 slf4j-api-1.6.1.jar	2015/8/6 14:05	Executable Jar File	25 KB
 slf4j-log4j12-1.7.2.jar	2015/8/6 14:05	Executable Jar File	9 KB




log4j 的包由 Spring 引入.

数据库驱动:

 mysql-connector-java-5.1.7-bin.jar	2014/7/8 18:41	Executable Jar File	694 KB
--	----------------	---------------------	--------





Hibernate 引入连接池:

D:\hibernate-release-5.0.7.Final\lib\optional\c3p0*.jar




 c3p0-0.9.2.1.jar	2014/4/28 20:30	Executable Jar File	414 KB
 hibernate-c3p0-5.0.7.Final.jar	2016/1/13 12:42	Executable Jar File	12 KB
 mchange-commons-java-0.2.3.4.jar	2014/4/28 20:30	Executable Jar File	568 KB







【Spring】

基本的开发:

 com.springsource.org.apache.commons.logging-1.1.1.jar
 com.springsource.org.apache.log4j-1.2.15.jar
 spring-beans-4.2.4.RELEASE.jar
 spring-context-4.2.4.RELEASE.jar
 spring-core-4.2.4.RELEASE.jar
 spring-expression-4.2.4.RELEASE.jar

AOP 开发:

 spring-aop-4.2.4.RELEASE.jar	2015/12/17 0:44	Executable Jar File	362 KB
 spring-aspects-4.2.4.RELEASE.jar	2015/12/17 0:48	Executable Jar File	58 KB
 com.springsource.org.aopalliance-1.0.0.jar	2010/4/2 11:09	Executable Jar File	5 KB

 com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar	2010/4/2 11:09	Executable Jar File	1,604 KB
JDBC 开发:			
 spring-tx-4.2.4.RELEASE.jar	2015/12/17 0:45	Executable Jar File	260 KB
 spring-jdbc-4.2.4.RELEASE.jar	2015/12/17 0:45	Executable Jar File	414 KB
事务管理的开发:			
 spring-tx-4.2.4.RELEASE.jar	2015/12/17 0:45	Executable Jar File	260 KB
整合 Hibernate:			
 spring-orm-4.2.4.RELEASE.jar	2015/12/17 0:46	Executable Jar File	456 KB
整合 web 项目:			
 spring-web-4.2.4.RELEASE.jar	2015/12/17 0:46	Executable Jar File	750 KB

1.2.2.2 引入相关的配置文件:

【Struts2】

```
web.xml
<!-- 配置 Struts2 的核心过滤器 -->
<filter>
    <filter-name>struts2</filter-name>

    <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
struts.xml
```

【Hibernate】

核心配置: hibernate.cfg.xml

映射文件:

【Spring】

```
web.xml
<!-- 配置 Spring 的核心监听器 -->
<listener>

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



```
class>
    </listener>

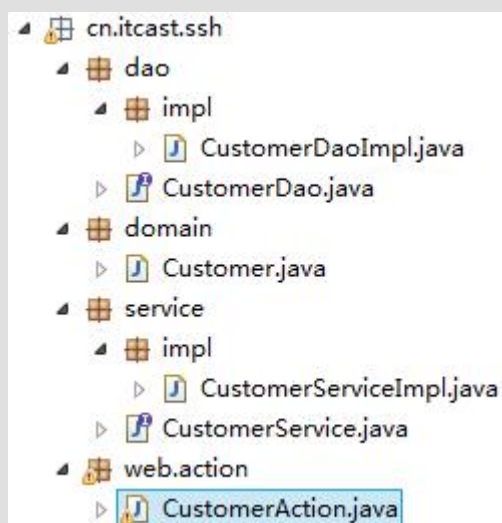
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>
applicationContext.xml

log4j.properties
```

1.2.2.3 引入相关的页面并进行修改:

```
<FORM id=form1 name=form1
    action="${pageContext.request.contextPath }/customer_save.action|"
```

1.2.2.4 创建包结构和相关的类:



```
cn.itcast.ssh
├── dao
│   ├── impl
│   │   └── CustomerDaoImpl.java
│   └── CustomerDao.java
├── domain
│   └── Customer.java
├── service
│   ├── impl
│   │   └── CustomerServiceImpl.java
│   └── CustomerService.java
└── web.action
    └── CustomerAction.java
```

1.2.2.5 Struts2 和 Spring 的整合:方式一: Action 类由 Struts2 自己创建

【编写 Action 中的 save 方法】

```
/**
 * 保存客户的执行的方法:save
 */
public String save() {
    System.out.println("Action 中的 save 方法执行了...");
    return NONE;
}
```

【配置 Action 类】

```
<package name="ssh" extends="struts-default" namespace="/">
    <action name="customer_*" class="cn.itcast.ssh.web.action.CustomerAction"
method="{1}">

        </action>
    </package>
```

【在 Action 中调用业务层的类】

```
配置 Service:
    <!-- 配置 Service -->
    <bean                                id="customerService"
class="cn.itcast.ssh.service.impl.CustomerServiceImpl">

        </bean>
```

在 Action 中调用

// 传统方式的写法

```
WebApplicationContext webApplicationContext = WebApplicationContextUtils
.getWebApplicationContext(ServletActionContext.getServletContext());
CustomerService customerService = (CustomerService)
webApplicationContext.getBean("customerService");
```

***** 这种写法很麻烦的，因为需要在每个 Action 中的每个方法上获取工厂，通过工厂获得类。

为了简化这个代码引入一个插件的包：

struts2-spring-plugin-2.3.24.jar

在这个插件中开启一个 Struts2 常量

```
* <constant name="struts.objectFactory" value="spring" />
* 默认的情况下 struts2 将这个常量关闭的，现在引入插件以后，将常量开启了，引发下面的一些
常量生效。
```

```
struts.objectFactory.spring.autoWire = name
```

那么就可以在 Action 中提供想注入的属性了：

```
public class CustomerAction extends ActionSupport implements ModelDriven<Customer>
{
    // 模型驱动使用的对象
    private Customer customer = new Customer();

    @Override
    public Customer getModel() {
        return customer;
    }

    // 注入业务层的类：
```



```
private CustomerService customerService;

public void setCustomerService(CustomerService customerService) {
    this.customerService = customerService;
}

/**
 * 保存客户的执行的方法:save
 */
public String save() {
    System.out.println("Action 中的 save 方法执行了...");
    // 传统方式的写法
    /*WebApplicationContext webApplicationContext = WebApplicationContextUtils
        .getWebApplicationContext(ServletActionContext.getServletContext());
    CustomerService customerService = (CustomerService)
        webApplicationContext.getBean("customerService");*/

    // 自动注入
    customerService.save(customer);
    return NONE;
}
}
```

【在 Service 中编写 save 方法】

```
public class CustomerServiceImpl implements CustomerService {

    @Override
    public void save(Customer customer) {
        System.out.println("Service 中的 save 方法执行了...");
    }

}
```

1.2.2.6 Struts2 和 Spring 的整合方式二:Action 类由 Spring 创建。(推荐)

【引入插件包】

```
struts2-spring-plugin-2.3.24.jar
```

【Action 交给 Spring 管理】

将 Action 配置到 Spring 中.

```
<!-- 配置 Action -->
```

```
<bean id="customerAction" class="cn.itcast.ssh.web.action.CustomerAction"
```

```
scope="prototype">
    <!--必须手动注入属性-->
    <property name="customerService" ref="customerService"/>
</bean>

Action 的配置:
<package name="ssh" extends="struts-default" namespace="/">
    <action name="customer_*" class="customerAction" method="{1}">

    </action>
</package>
```

1.2.2.7 在业务层调用 DAO

【将 DAO 配置到 Spring 中】

```
<!-- 配置 DAO -->
<bean id="customerDao" class="cn.itcast.ssh.dao.impl.CustomerDaoImpl">

</bean>
```

【在业务层注入 Dao】

```
public class CustomerServiceImpl implements CustomerService {

    // 注入 Dao
    private CustomerDao customerDao;

    public void setCustomerDao(CustomerDao customerDao) {
        this.customerDao = customerDao;
    }

    @Override
    public void save(Customer customer) {
        System.out.println("Service 中的 save 方法执行了...");
        customerDao.save(customer);
    }

}

<!-- 配置 Service -->
<bean
class="cn.itcast.ssh.service.impl.CustomerServiceImpl"
    <property name="customerDao" ref="customerDao"/>
</bean>
```

1.2.2.8 Spring 整合 Hibernate:

【创建映射文件】

```
<hibernate-mapping>

    <class name="cn.itcast.ssh.domain.Customer" table="cst_customer">

        <id name="cust_id">
            <generator class="native"/>
        </id>

        <property name="cust_name"/>
        <property name="cust_user_id"/>
        <property name="cust_create_id"/>
        <property name="cust_source"/>
        <property name="cust_industry"/>
        <property name="cust_level"/>
        <property name="cust_linkman"/>
        <property name="cust_phone"/>
        <property name="cust_mobile"/>

    </class>
</hibernate-mapping>
```

【加载到核心配置文件】

```
<!-- 加载映射文件 -->
<mapping resource="cn/itcast/ssh/domain/Customer.hbm.xml"/>
```

【在 Spring 的配置文件中完成如下配置】

```
<!-- 配置 Hibernate 中的 sessionFactory -->
<bean id="sessionFactory"
class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="configLocations" value="classpath:hibernate.cfg.xml"/>
</bean>
```

【改写 DAO】

```
public class CustomerDaoImpl extends HibernateDaoSupport implements CustomerDao {

    @Override
    public void save(Customer customer) {
        System.out.println("DAO 中的 save 方法执行了...");
    }

}

<!-- 配置 DAO -->
```

```
<bean id="customerDao" class="cn.itcast.ssh.dao.impl.CustomerDaoImpl">
  <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

【调用模板中的方法】

```
@Override
public void save(Customer customer) {
    System.out.println("DAO 中的 save 方法执行了...");
    // 保存:
    this.getHibernateTemplate().save(customer);
}
```

1.2.2.9 配置 Spring 的事务管理：

【配置事务管理器】

```
<!-- 配置事务管理器 -->
<bean id="transactionManager"
class="org.springframework.orm.hibernate5.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

【注解事务管理的开启】

```
<!-- 开启事务管理的注解 -->
<tx:annotation-driven transaction-manager="transactionManager"/>
```

【在业务层添加一个注解】

```
@Transactional
public class CustomerServiceImpl implements CustomerService {
```

1.2.3 SSH 框架的整合方式二：不带 Hibernate 的配置文件

1.2.3.1 复制一个 SSH1 的项目。

1.2.3.2 查看 Hibernate 的配置文件：

Hibernate 的配置文件包含如下内容：

- 连接数据库必要的参数：
- Hibernate 的属性：
- 连接池的配置：
- 映射文件的引入：

1.2.3.3 替换数据库连接参数和连接池的配置:

创建 jdbc.properties

```
jdbc.driverClass=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql:///ssh1
jdbc.username=root
jdbc.password=123
```

在 Spring 中引入外部属性文件

```
<!-- 引入外部属性文件 -->
```

```
<context:property-placeholder location="classpath:jdbc.properties"/>
```

配置连接池:

```
<!-- 配置 c3p0 连接池: -->
```

```
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
  <property name="driverClass" value="{jdbc.driverClass}"/>
  <property name="jdbcUrl" value="{jdbc.url}"/>
  <property name="user" value="{jdbc.username}"/>
  <property name="password" value="{jdbc.password}"/>
</bean>
```

1.2.3.4 配置 Hibernate 的其他属性及映射:

```
<!-- 配置 Hibernate 中的 sessionFactory -->
```

```
<bean id="sessionFactory"
```

```
class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
```

```
<!-- 注入连接池 -->
```

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

```
<!-- 配置 Hibernate 的相关属性 -->
```

```
<property name="hibernateProperties">
```

```
<props>
```

```
<!-- 配置 Hibernate 的方言 -->
```

```
<prop
```

```
key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
```

```
<!-- 显示 SQL -->
```

```
<prop key="hibernate.show_sql">true</prop>
```

```
<!-- 格式化 SQL -->
```

```
<prop key="hibernate.format_sql">true</prop>
```

```
<!-- 映射到 DDL 的自动创建 -->
```

```
<prop key="hibernate.hbm2ddl.auto">update</prop>
```

```
</props>
```

```
</property>
```

```
<!-- 配置引入映射文件 -->
```

```
<property name="mappingResources">
    <list>
        <value>cn/itcast/ssh/domain/Customer.hbm.xml</value>
    </list>
</property>
</bean>
```

1.2.4 HibernateTemplate 的使用:

```
public class CustomerDaoImpl extends HibernateDaoSupport implements CustomerDao {

    @Override
    public void save(Customer customer) {
        System.out.println("DAO 中的 save 方法执行了...");
        // 保存:
        this.getHibernateTemplate().save(customer);
    }

    @Override
    public void update(Customer customer) {
        this.getHibernateTemplate().update(customer);
    }

    @Override
    public void delete(Customer customer) {
        this.getHibernateTemplate().delete(customer);
    }

    @Override
    public Customer findById(Long id) {
        return this.getHibernateTemplate().load(Customer.class, id);
    }

    @Override
    public List<Customer> findAllByHQL() {
        List<Customer> list = (List<Customer>)
        this.getHibernateTemplate().find("from Customer");
        return list;
    }

    public List<Customer> findAllByQBC() {
        DetachedCriteria detachedCriteria =
        DetachedCriteria.forClass(Customer.class);
        List<Customer> list = (List<Customer>)
```

```
this.getHibernateTemplate().findByCriteria(detachedCriteria);  
    return list;  
}  
  
}
```

1.2.5 延迟加载的问题的解决：OpenSessionInViewFilter

```
<filter>  
    <filter-name>OpenSessionInViewFilter</filter-name>  
  
    <filter-class>org.springframework.orm.hibernate5.support.OpenSessionInViewFilter</filter-class>  
</filter>  
  
<filter-mapping>  
    <filter-name>OpenSessionInViewFilter</filter-name>  
    <url-pattern>*.action</url-pattern>  
</filter-mapping>
```