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

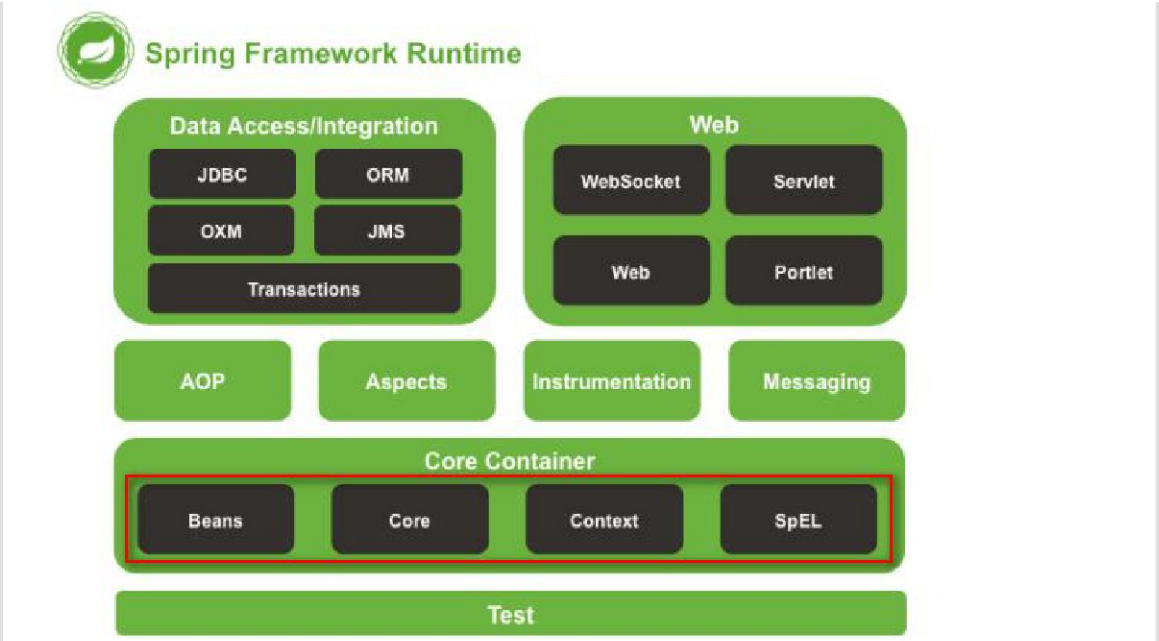
1.2.1 Spring 的 Bean 管理:(注解方式)

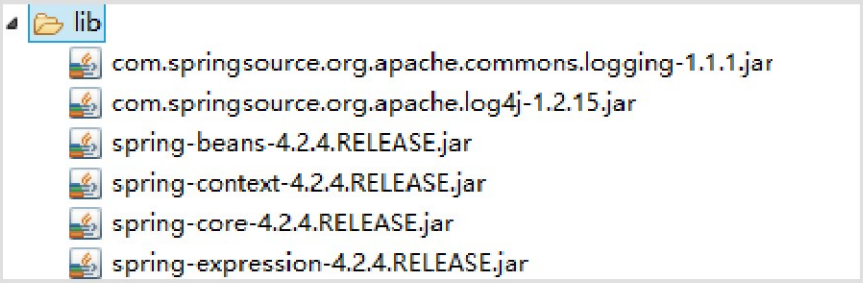
1.2.1.1 步骤一:下载 Spring 的开发包:

```
    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:





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-configur
ation.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 管理的中常用的注解:

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:

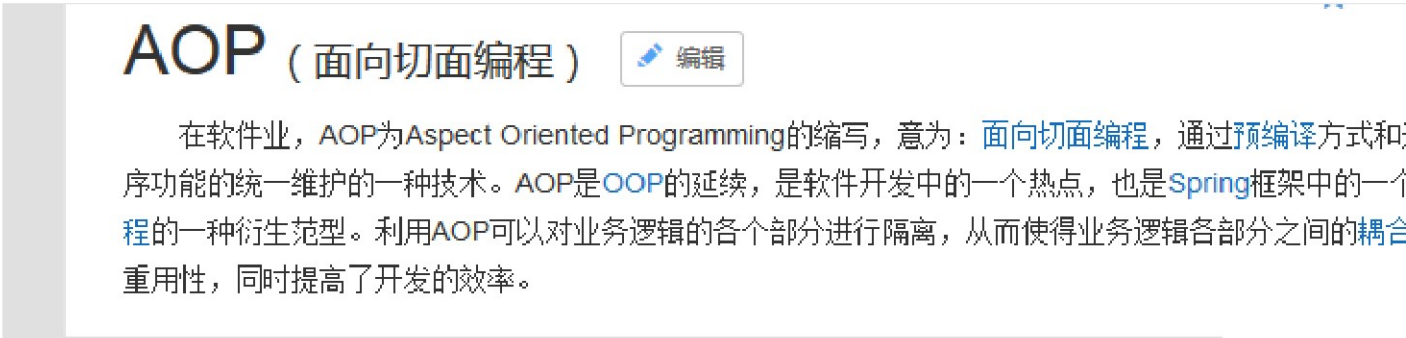
* XML :结构清晰.
* 注解 :开发方便.(属性注入.)

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
序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
程的一种衍生范型。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
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 = (UserDao) 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在期织入

Proxy（代理）:一个类被 AOP 织入增强后，就产生一个结果代理类

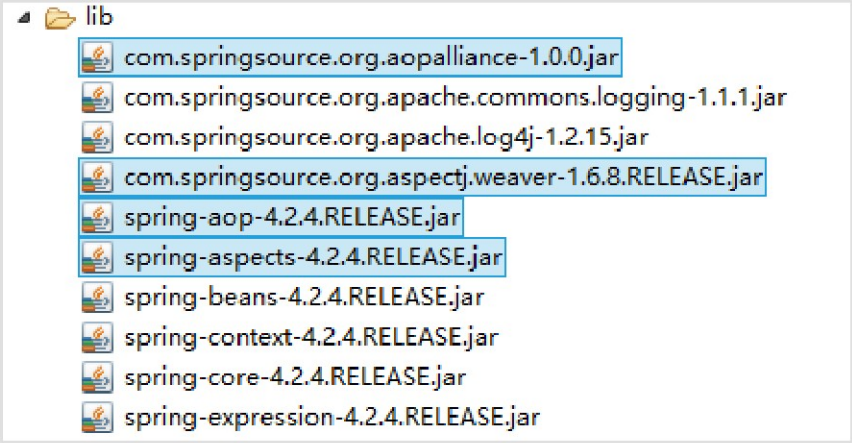
Aspect(切面): 是切入点和通知（引介）的结合

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>
```