Spring学习日志二

一、使用注解代替xml配置

导包略

1.1 注解配置

先导入新的命名空间
并开启注解配置

```
<!-- 指定扫描cn.scct.bean下的所有类的注解（递归扫描） -->
<context:component-scan base-package="cn.scct.bean"></context:component-scan>
```

上面的意思是`cn.scct.bean`下的所有类都开启注解。

1.2 使用注解注册bean

```
@Component("user")
//相当于<bean name="user" class="cn.itcast.bean.User"  />
//@Service("user") // service层
//@Controller("user") // web层
//@Repository("user")// dao层
```

但其实，他们没有任何区别，和@Component，除了能给人区分作用而已，
@Component添加到类的定义前，表示注册到容器中
同时，@Service @Controller @Repository分别表示注册到各个层
在任何层都能用

1.3 使用注解代替xml中bean的scope属性

```
//指定对象的作用范围
@Scope(scopeName="singleton")//默认就是单例的
```

类定义前

1.4 值类型属性注入

```
19 public class User {
20     private String name;
21     @Value("18")
22     private Integer age;
```

//值类型注入　会破坏封装性

```
@Value("tom")
public void setName(String name) {
    this.name = name;
}
```

方法二

//在set方法前进行注入，推荐使用

1.5 引用类型属性注入

注意自动注入@Autowired，ssm项目中常用

```
//@Autowired //自动装配
//问题：如果匹配多个类型一致的对象，将无法选择具体注入哪一个对象。
//@Qualifier("car2")//使用@Qualifier注解告诉spring容器自动装配哪个名称的对象

@Resource(name="car")//手动注入，指定注入哪个名称的对象
private Car car;
```

这个最好

```
<!-- <bean name="car2" class="cn.scct.bean.Car"  >
    <property name="name" value="布加迪威龙" ></property>
    <property name="color" value="black"  ></property>
</bean> -->
```

这里在配置文件中配置了一个car类型，自动装配无法区分那个对象被注入

1.6 初始化/销毁方法

```
@PostConstruct //在对象被创建后调用.init-method
public void init(){
    System.out.println("我是初始化方法!");
}
@PreDestroy //在销毁之前调用.destory-method
public void destory(){
    System.out.println("我是销毁方法!");
}
```

二、spring整合junit测试

1.导包4+2+aop+test

2.配置注解

```
//帮我们创建容器
@RunWith(SpringJUnit4ClassRunner.class)
//指定创建容器时使用哪个配置文件
@ContextConfiguration("classpath:applicationContext.xml")
public class Demo {
    //将名为user的对象注入到u变量中
    @Resource(name="user")
    private User u;
```

3.测试

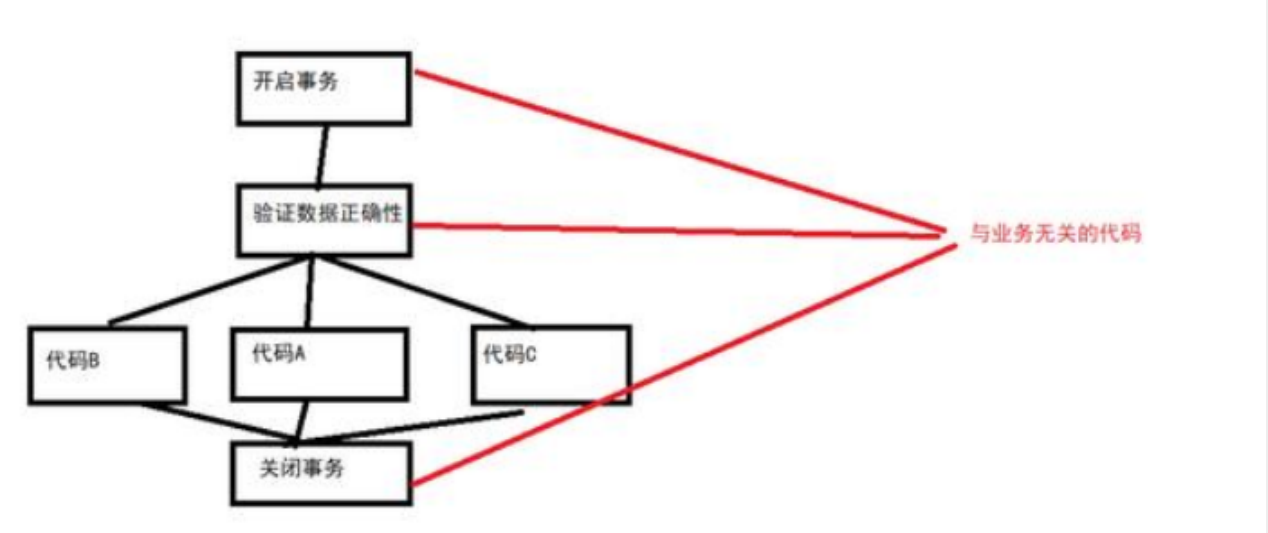
```
@Test
public void fun1(){

    System.out.println(u);

}
```

因为我们测试的时候老是要创建容器，配置文件等。

三、aop思想

3.1 aop思想介绍

AOP(Aspect-Oriented Programming,面向切面编程)，可以说是OOP(Object-Oriented Programming,面向切面编程)的补充和完善。OOP引入封装、继承和多态性来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为时，OOP则显得无能为力。也就是说OOP允许你定义从上到下的纵向关系，但并不适合定义从左到右的横向关系。例如日志功能。日志代码往往水平的散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。对于其他类型的代码，如安全性、异常处理和透明的持续性也是如此，它导致了大量代码的重复，而不利于各个模块的重用。
而AOP技术相反，它利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其命名为“Aspect”，即切面。所谓的切面，就是将那些与业务无关，却为业务模块所共同调用的逻辑或者责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有益于未来的可操作性和可维护性。AOP代表的是一个横向关系。如果说对象是一个空心的圆柱体，其中封装的是对象的属性和行为；那么面向切面编程的方法，仿佛一把利刃，将这个空心圆柱体剖开，以获取其内部的消息。

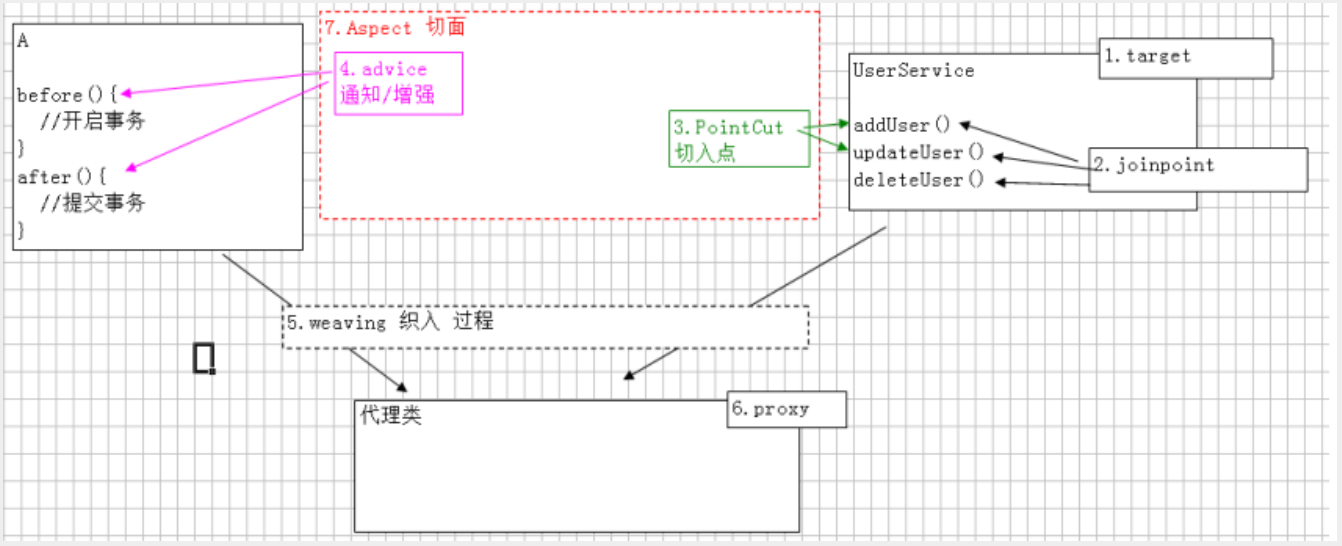


使用"横切"技术，AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分就是横切关注点。横切关注点的一个特点就是，他们经常发生在核心关注点的多处，而各处都基本相似。比如权限认证、日志、事务处理。AOP的作用在于分离系统的各种关注点，将核心关注点和横切关注点分离开来。正如Avanade公司的高级方案架构师Adam Magee所说，AOP的核心思想就是"将应用程序中的商业逻辑同对其提供支持的通用服务进行分离。"

四、aop名词解释

1. **target(目标类)**：需要被代理的类。例如：UserService
2. **Joinpoint(连接点)**：所谓连接点是指那些可能被拦截到的方法。例如：所有的方法
3. **PointCut(切入点)**：已经被增强的连接点。例如：addUser()
4. **advice(通知/增强，增强代码)**。例如：after、before
5. **Weaving(织入)**：是指把增强advice应用到目标对象target来创建新的代理对象proxy的过程.
6. **proxy(代理类)**

7. **Aspect(切面)**：是切入点pointcut和通知advice的结合。 一个线是一个特殊的面。 一个切入点和一个通知，组成成一个特殊的面。



五、aop实现原理

5.1 aop实现原理一：JDK动态代理

JDK动态代理 对"装饰者"设计模式 简化。使用前提：必须有接口

1. 目标类：接口 + 实现类
2. 切面类：用于存通知 MyAspect
3. 工厂类：编写工厂生成代理
4. 测试

5.1.1 目标类

```java
public interface UserService {
    public void addUser();
    public void updateUser();
    public void deleteUser();
}
```

```java
public class UserServiceImpl implements UserService {

    @Override
    public void addUser() {
        // TODO Auto-generated method stub
        System.out.println("add方法");
    }

    @Override
    public void updateUser() {
        // TODO Auto-generated method stub
         System.out.println("update方法");
    }

    @Override
    public void deleteUser() {
        // TODO Auto-generated method stub
         System.out.println("delete方法");
    }

}
```

5.1.2 切面类

```java
public class MyAspect {
    public void before(){
        System.out.println("不做鸡头");
    }
    public void after(){
        System.out.println("宁做凤尾");
    }

}
```

5.1.3 工厂类

```java
public class MyBeanFactory {
    public static UserService createService(){
        //1 目标类
        final UserService userService = new UserServiceImpl();
        //2切面类
        final MyAspect myAspect = new MyAspect();
        /* 3 代理类：将目标类（切入点）和 切面类（通知） 结合 --> 切面
         *  Proxy.newProxyInstance
         *      参数1：loader ，类加载器，动态代理类 运行时创建，任何类都需要类加载器将其加载到内存。
         *          一般情况：当前类.class.getClassLoader();
         *                   目标类实例.getClass().get...
         *      参数2：Class[] interfaces 代理类需要实现的所有接口
         *          方式1：目标类实例.getClass().getInterfaces()  ;注意：只能获得自己接口，不能获得父元素接口
         *          方式2：new Class[]{UserService.class}
         *          例如：jdbc 驱动 --> DriverManager  获得接口 Connection
         *      参数3：InvocationHandler  处理类，接口，必须进行实现类，一般采用匿名内部
         *          提供 invoke 方法，代理类的每一个方法执行时，都将调用一次invoke
         *              参数31：Object proxy ：代理对象
         *              参数32：Method method ： 代理对象当前执行的方法的描述对象（反射）
         *                  执行方法名：method.getName()
         *                  执行方法：method.invoke(对象，实际参数)
         *              参数33：Object[] args ：方法实际参数
         *
         */
        UserService proxService = (UserService)Proxy.newProxyInstance(
                                  MyBeanFactory.class.getClassLoader(),
                                  userService.getClass().getInterfaces(),
                                  new InvocationHandler() {
                                      @Override
                                      public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                                          //前执行
                                          myAspect.before();
                                          //执行目标类的方法
                                          Object obj = method.invoke(userService, args);
                                          //后执行
                                          myAspect.after();
                                          return obj;
                                      }
                                  });

        return proxService;
    }
}
```

5.1.4 测试

```java
public class TestJDK {
    @Test
    public void test1(){
        UserService userService = MyBeanFactory.createService();
        userService.addUser();
        userService.updateUser();
        userService.deleteUser();
    }

}
```

```
不做鸡头
add方法
宁做凤尾
不做鸡头
update方法
宁做凤尾
不做鸡头
delete方法
宁做凤尾
```

5.2 aop实现原理二：cglib代理

没有接口，只有实现类。 采用字节码增强框架cglib，在运行时创建目标类的子类，从而对目标类进行增强。
切面类和目标实现类都是一样，主要看工厂类结合着两者的代码。

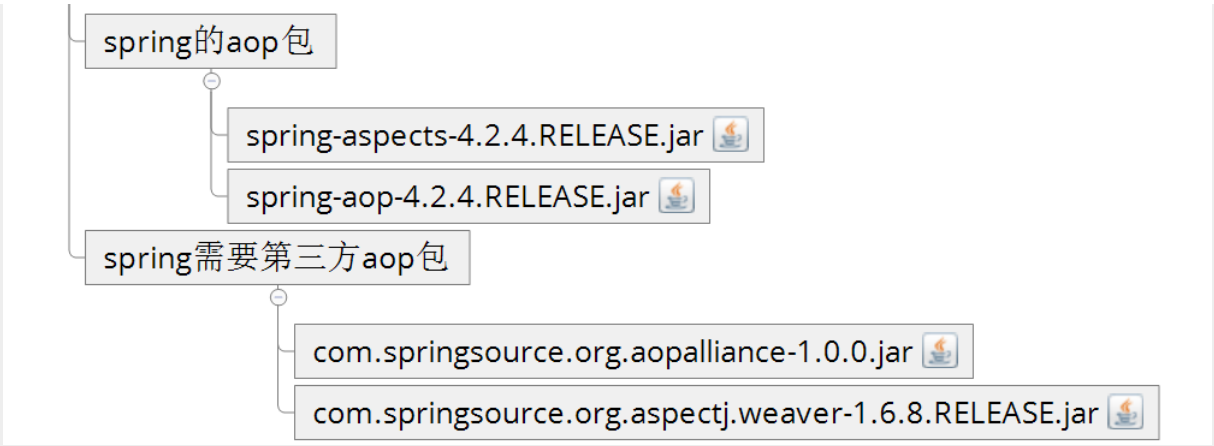
工厂类实现

```
public class MyBeanFactory {
    public static UserServiceImpl createService(){
        //1 目标类
        final UserServiceImpl userService = new UserServiceImpl();
        //2切面类
        final MyAspect myAspect = new MyAspect();
        // 3.代理类，采用cglib，底层创建目标类的子类
        //3.1 核心类
        Enhancer enhancer = new Enhancer();
        //3.2 确定父类
        enhancer.setSuperclass(userService.getClass());

        /* 3.3 设置回调函数， MethodInterceptor接口 等效 jdk InvocationHandler接口
         *  intercept() 等效 jdk  invoke()
         *       参数1、参数2、参数3：以invoke一样
         *       参数4：methodProxy 方法的代理
         */
        enhancer.setCallback(new MethodInterceptor(){

            @Override
            public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {

                //前
                myAspect.before();

                //执行目标类的方法
                Object obj = method.invoke(userService, args);
                // * 执行代理类的父类，执行目标类（目标类和代理类 父子关系）
                //methodProxy.invokeSuper(proxy, args);

                //后
                myAspect.after();

                return obj;
            }
        });
        //3.4 创建代理
        UserServiceImpl proxService = (UserServiceImpl) enhancer.create();

        return proxService;
    }
}
```

六、spring中的aop演示

6.1 配置文件版本

6.1.1 导包（额外）



6.1.2 准备目标对象

```java
package cn.scct.b_target;
public class UserServiceImpl implements UserService {
    @Override
    public void save() {
        System.out.println("保存用户!");
        //int i = 1/0;
    }
    @Override
    public void delete() {
        System.out.println("删除用户!");
    }
    @Override
    public void update() {
        System.out.println("更新用户!");
    }
    @Override
    public void find() {
        System.out.println("查找用户!");
    }
}
```

6.1.3 准备通知

```java
//通知类
public class MyAdvice {

    //前置通知
//   |-目标方法运行之前调用
//后置通知(如果出现异常不会调用)
//   |-在目标方法运行之后调用
//环绕通知
//   |-在目标方法之前和之后都调用
//异常拦截通知
//   |-如果出现异常,就会调用
//后置通知(无论是否出现 异常都会调用)
//   |-在目标方法运行之后调用
//-----------------------------------------------------------------
//前置通知
public void before(){
    System.out.println("这是前置通知!!");
}
//后置通知
public void afterReturning(){
    System.out.println("这是后置通知(如果出现异常不会调用)!!");
}
//环绕通知
public Object around(ProceedingJoinPoint pjp) throws Throwable {
    System.out.println("这是环绕通知之前的部分!!");
    Object proceed = pjp.proceed();//调用目标方法
    System.out.println("这是环绕通知之后的部分!!");
    return proceed;
}
//异常通知
public void afterException(){
    System.out.println("出事啦!出现异常了!!");
}
//后置通知
public void after(){
    System.out.println("这是后置通知(出现异常也会调用)!!");
}
}
```

6.1.4 配置

```xml
<!-- 准备工作: 导入aop(约束)命名空间 -->
<!-- 1.配置目标对象 -->
    <bean name="userService" class="cn.scct.b_target.UserServiceImpl" ></bean>
<!-- 2.配置通知对象 -->
    <bean name="myAdvice" class="cn.scct.d_springaop.MyAdvice" ></bean>
<!-- 3.配置将通知织入目标对象 -->
    <aop:config>
        <!-- 配置切入点
        切入点表达式
            public void cn.scct.b_target.UserServiceImpl.save() //只为该方法配置切入点
            void cn.scct.b_target.UserServiceImpl.save()//默认是public,所以默认不写
            * cn.scct.b_target.UserServiceImpl.save()//对返回值不对类求，写*
            * cn.scct.b_target.UserServiceImpl.*()//为这个类的所有空参方法进行切入

            * cn.scct.b_target.*ServiceImpl.*(..)//找该包下的以ServiceImpl结尾的类的所有方法
            * cn.scct.b_target..*ServiceImpl.*(..)//找该包下及其子包的以ServiceImpl结尾的类的所有方法
        -->
        <aop:pointcut expression="execution(* cn.scct.b_target.*ServiceImpl.*(..))" id="pc"/>
        <aop:aspect ref="myAdvice" >
            <!-- 指定名为before方法作为前置通知 -->
            <aop:before method="before" pointcut-ref="pc" />
            <!-- 后置 -->
            <aop:after-returning method="afterReturning" pointcut-ref="pc" />
            <!-- 环绕通知 -->
            <aop:around method="around" pointcut-ref="pc" />
            <!-- 异常拦截通知 -->
            <aop:after-throwing method="afterException" pointcut-ref="pc"/>
            <!-- 后置 -->
            <aop:after method="after" pointcut-ref="pc"/>
        </aop:aspect>
    </aop:config>
</beans>
```

被

表示切入点 可以增强的方法 来自于目标类的方法

来自目标类

来自通知类(切面类)

表示通知，增强的方法

随便取的名字

各种通知，拼命加上，以演示

6.1.5 测试

这里是用注解测试的

```java
import javax.annotation.Resource;
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:cn/scct/d_springaop/applicationContext.xml")
public class Demo {
    @Resource(name="userService")
    private UserService us;

    @Test
    public void fun1(){
        us.save();
    }

}
```

```
这是前置通知!!
这是环绕通知之前的部分!!
保存用户!
这是后置通知(出现异常也会调用)!!
这是环绕通知之后的部分!!
这是后置通知(如果出现异常不会调用)!!
```

6.2 注解版本

首先在配置文件中开启注解以便完成织入

```xml
<!-- 准备工作: 导入aop(约束)命名空间 -->
<!-- 1.配置目标对象 -->
    <bean name="userService" class="cn.scct.b_target.UserServiceImpl" ></bean>
<!-- 2.配置通知对象 -->
     <bean name="myAdvice" class="cn.scct.e_springaop.MyAdvice" ></bean>

<!-- 3.开启使用注解完成织入 -->
     <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
```

目标类不用变

通知类进行注解织入

```java
//通知类
@Aspect          //表示通知类
//表示该类是一个通知类
public class MyAdvice {
    @Pointcut("execution(* cn.itcast.service.*ServiceImpl.*(..))")
    public void pc(){}                    //借个名字而已，创建一个方法表示切入点表达式
    //前置通知
    //指定该方法是前置通知,并制定切入点
    @Before("MyAdvice.pc()") //然后这里可以直接调用该方法得到切入点表达式
    public void before(){
        System.out.println("这是前置通知!!");
    }
    //后置通知                    //不厌其烦的写法就是这样，直接在里面写表达式
    @AfterReturning("execution(* cn.itcast.service.*ServiceImpl.*(..))")
    public void afterReturning(){
        System.out.println("这是后置通知(如果出现异常不会调用)!!");
    }
    //环绕通知
    @Around("execution(* cn.itcast.service.*ServiceImpl.*(..))")
    public Object around(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("这是环绕通知之前的部分!!");
        Object proceed = pjp.proceed();//调用目标方法
        System.out.println("这是环绕通知之后的部分!!");
        return proceed;
    }
    //异常通知
    @AfterThrowing("execution(* cn.itcast.service.*ServiceImpl.*(..))")
    public void afterException(){
        System.out.println("出事啦!出现异常了!!");
    }
    //后置通知
    @After("execution(* cn.itcast.service.*ServiceImpl.*(..))")
    public void after(){
        System.out.println("这是后置通知(出现异常也会调用)!!");
    }
}
```

测试

```java
import javax.annotation.Resource;
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:cn/scct/e_springaop/applicationContext.xml")
public class Demo {
    @Resource(name="userService")
    private UserService us;

    @Test
    public void fun1(){
        us.save();
    }

}
```

```
这是环绕通知之前的部分!!
这是前置通知!!
保存用户!
这是环绕通知之后的部分!!
这是后置通知(出现异常也会调用)!!
这是后置通知(如果出现异常不会调用)!!
```