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。

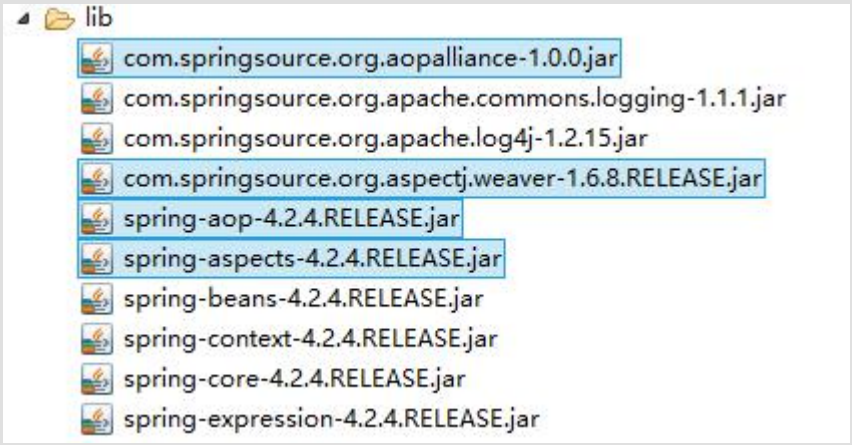
1.2 相关知识点

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:

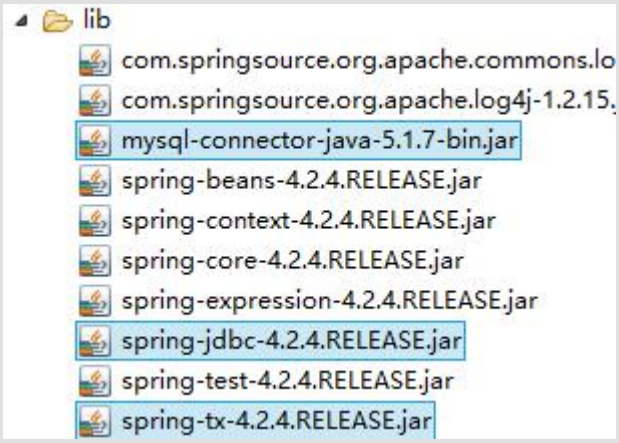
1.4.1.1 Spring 提供了很多持久层技术的模板类简化编程:

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】

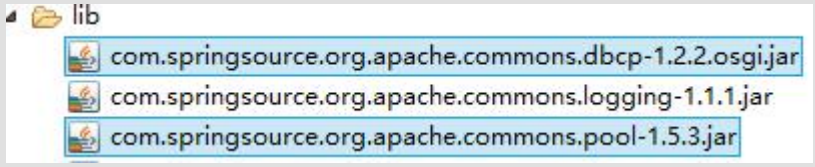
```
<!-- 配置 JDBC 模板 -->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
 <property name="dataSource" ref="dataSource"/>
</bean>
```

【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.demo1.AccountServiceImpl">
         <property name="accountDao" ref="accountDao"/>
        </bean>

        <!-- 配置 DAO 的类 -->
        <bean id="accountDao" class="cn.itcast.transaction.demo1.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
```

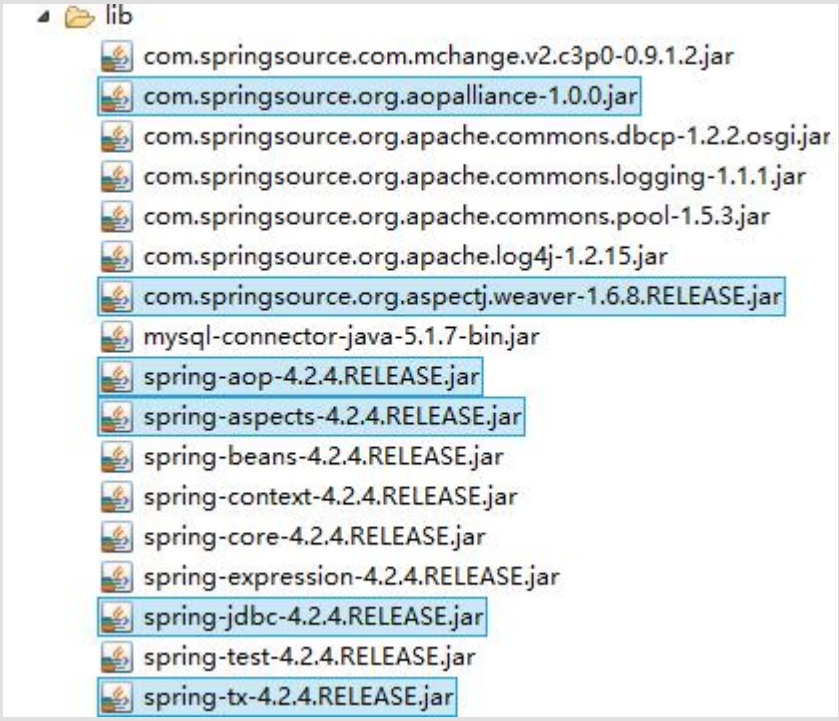
```
                read-only="false"   只读
                timeout="-1"        过期时间
                rollback-for=""     -Exception
                no-rollback-for=""  +Exception
            -->
        <tx:method name="transfer" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
```

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 {
```