## AOP基于注解

### S 引入相关的jar包:

\* spring的传统AOP的开发的包

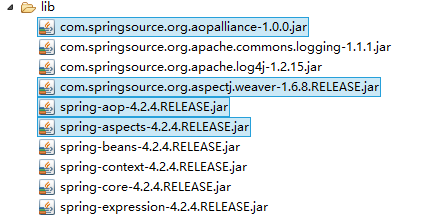
spring-aop-4.2.4.RELEASE.jar

com.springsource.org.aopalliance-1.0.0.jar

\* aspectJ的开发包:

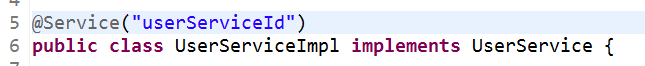
com.springsource.org.aspectj.weaver-1.6.8.RELEASE.jar

spring-aspects-4.2.4.RELEASE.jar



### 替换bean

|  |
| --- |
| <!-- 1 创建目标类 -->  <bean id=*"userServiceId"* class=*"com.qf.d\_aspect.b\_anno.UserServiceImpl"*></bean>  <!-- 2 创建切面类（通知） -->  <bean id=*"myAspectId"* class=*"com.qf.d\_aspect.b\_anno.MyAspect"*></bean> |





* 注意：扫描

|  |
| --- |
| <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"*  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*  *http://www.springframework.org/schema/context*  *http://www.springframework.org/schema/context/spring-context.xsd"*>  <!-- 1.扫描 注解类 -->  <context:component-scan base-package=*"com.qf.d\_aspect.b\_anno"*></context:component-scan> |

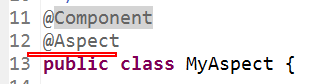
### 替换aop

* 必须进行aspectj 自动代理

|  |
| --- |
| <!-- 2.确定 aop注解生效 -->  <aop:aspectj-autoproxy></aop:aspectj-autoproxy> |

* 声明切面

|  |
| --- |
| <aop:aspect ref=*"myAspectId"*> |



* 替换前置通知

|  |
| --- |
| <aop:before method=*"myBefore"* pointcut=*"execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))"*/> |

|  |
| --- |
| //切入点当前有效  @Before("execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))")  **public** **void** myBefore(JoinPoint joinPoint){  System.*out*.println("前置通知 ： " + joinPoint.getSignature().getName());  } |

* 替换 公共切入点

|  |
| --- |
| <aop:pointcut expression=*"execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))"* id=*"myPointCut"*/> |

|  |
| --- |
| //声明公共切入点  @Pointcut("execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))")  **private** **void** myPointCut(){  } |

* 替换后置

|  |
| --- |
| <aop:after-returning method=*"myAfterReturning"* pointcut-ref=*"myPointCut"* returning=*"ret"* /> |

|  |
| --- |
| @AfterReturning(value="myPointCut()" ,returning="ret")  **public** **void** myAfterReturning(JoinPoint joinPoint,Object ret){  System.*out*.println("后置通知 ： " + joinPoint.getSignature().getName() + " , -->" + ret);  } |

* 替换环绕

|  |
| --- |
| <aop:around method=*"myAround"* pointcut-ref=*"myPointCut"*/> |

|  |
| --- |
| @Around(value = "myPointCut()")  **public** Object myAround(ProceedingJoinPoint joinPoint) **throws** Throwable{  System.*out*.println("前");  //手动执行目标方法  Object obj = joinPoint.proceed();    System.*out*.println("后");  **return** obj;  } |

* 替换抛出异常

|  |
| --- |
| <aop:after-throwing method=*"myAfterThrowing"* pointcut=*"execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))"* throwing=*"e"*/> |

|  |
| --- |
| @AfterThrowing(value="execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))" ,throwing="e")  **public** **void** myAfterThrowing(JoinPoint joinPoint,Throwable e){  System.*out*.println("抛出异常通知 ： " + e.getMessage());  } |

### 切面类

|  |
| --- |
| /\*\*  \* 切面类，含有多个通知  \*/  @Component  @Aspect  **public** **class** MyAspect {    //切入点当前有效  // @Before("execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))")  **public** **void** myBefore(JoinPoint joinPoint){  System.*out*.println("前置通知 ： " + joinPoint.getSignature().getName());  }    //声明公共切入点  @Pointcut("execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))")  **private** **void** myPointCut(){  }    // @AfterReturning(value="myPointCut()" ,returning="ret")  **public** **void** myAfterReturning(JoinPoint joinPoint,Object ret){  System.*out*.println("后置通知 ： " + joinPoint.getSignature().getName() + " , -->" + ret);  }    // @Around(value = "myPointCut()")  **public** Object myAround(ProceedingJoinPoint joinPoint) **throws** Throwable{  System.*out*.println("前");  //手动执行目标方法  Object obj = joinPoint.proceed();    System.*out*.println("后");  **return** obj;  }    // @AfterThrowing(value="execution(\* com.qf.d\_aspect.b\_anno.UserServiceImpl.\*(..))" ,throwing="e")  **public** **void** myAfterThrowing(JoinPoint joinPoint,Throwable e){  System.*out*.println("抛出异常通知 ： " + e.getMessage());  }    @After("myPointCut()")  **public** **void** myAfter(JoinPoint joinPoint){  System.*out*.println("最终通知");  }  } |

### spring配置

|  |
| --- |
| <!-- 1.扫描 注解类 -->  <context:component-scan base-package=*"com.qf.d\_aspect.b\_anno"*></context:component-scan>    <!-- 2.确定 aop注解生效 -->  <aop:aspectj-autoproxy></aop:aspectj-autoproxy> |

### aop注解总结

@Aspect 声明切面，修饰切面类，从而获得 通知。

通知

@Before 前置

@AfterReturning 后置

@Around 环绕

@AfterThrowing 抛出异常

@After 最终

切入点

@PointCut ，修饰方法 private void xxx(){} 之后通过“方法名”获得切入点引用

# Spring框架的JDBC模板技术

## Spring框架的JDBC模板技术概述

**1. Spring框架中提供了很多持久层的模板类来简化编程，使用模板类编写程序会变的简单**

**2. 提供了JDBC模板，Spring框架提供的**

**JdbcTemplate类**

**3. Spring框架可以整合Hibernate框架，也提供了模板类**

**HibernateTemplate类**

## JDBC的模板类

**1. 步骤一：创建数据库的表结构**

create database spring\_day03;

use spring\_day03;

create table t\_account(

id int primary key auto\_increment,

name varchar(20),

money double

);

**2. 引入开发的jar包**

先引入IOC基本的6个jar包

再引入Spring-aop的jar包

最后引入JDBC模板需要的jar包

MySQL数据库的驱动包

Spring-jdbc.jar

Spring-tx.jar

**3. 编写测试代码（自己来new对象的方式）**

@Test

public void run1(){

// 创建连接池，先使用Spring框架内置的连接池

DriverManagerDataSource dataSource = new DriverManagerDataSource();

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql:///spring\_day03");

dataSource.setUsername("root");

dataSource.setPassword("root");

// 创建模板类

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

// 完成数据的添加

jdbcTemplate.update("insert into t\_account values (null,?,?)", "测试",10000);

}

## 使用Spring框架来管理模板类

**1. 刚才编写的代码使用的是new的方式，应该把这些类交给Spring框架来管理。**

2. 修改的步骤如下

**步骤一：Spring管理内置的连接池**

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql:///spring\_day03"/>

<property name="username" value="root"/>

<property name="password" value="root"/>

</bean>

步骤二：Spring管理模板类

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">

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

</bean>

步骤三：编写测试程序

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration("classpath:applicationContext.xml")

public class Demo2 {

@Resource(name="jdbcTemplate")

private JdbcTemplate jdbcTemplate;

@Test

public void run2(){

jdbcTemplate.update("insert into t\_account values (null,?,?)", "测试2",10000);

}

}

## Spring框架管理开源的连接池

1. 管理DBCP连接池

先引入DBCP的2个jar包

com.springsource.org.apache.commons.dbcp-1.2.2.osgi.jar

com.springsource.org.apache.commons.pool-1.5.3.jar

编写配置文件

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>

<property name="url" value="jdbc:mysql:///spring\_day03"/>

<property name="username" value="root"/>

<property name="password" value="root"/>

</bean>

2. 管理C3P0连接池

先引入C3P0的jar包

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

\* 编写配置文件

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="com.mysql.jdbc.Driver"/>

<property name="jdbcUrl" value="jdbc:mysql:///spring\_day03"/>

<property name="user" value="root"/>

<property name="password" value="root"/>

</bean>

## Spring框架的JDBC模板的简单操作

**1. 增删改查的操作**

@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,?,?)", "二狗子",1000d);

}

@Test

// 修改操作

public void demo2(){

jdbcTemplate.update("update account set name=?,money =? where id = ?", "jack",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 BeanMapper(), 1);

System.out.println(account);

}

@Test

// 查询所有记录

public void demo5(){

List<Account> list = jdbcTemplate.query("select \* from t\_account", new BeanMapper());

for (Account account : list) {

System.out.println(account);

}

}

}

class BeanMapper implements RowMapper<Account>{

public Account mapRow(ResultSet rs, int arg1) throws SQLException {

Account account = new Account();

account.setId(rs.getInt("id"));

account.setName(rs.getString("name"));

account.setMoney(rs.getDouble("money"));

return account;

}

}

# Spring框架的事务管理

## 事务的回顾

1. 事务：指的是逻辑上一组操作，组成这个事务的各个执行单元，要么一起成功,要么一起失败！

2. 事务的特性 ACID

\* 原子性

\* 一致性

\* 隔离性

\* 持久性

3. 如果不考虑隔离性,引发安全性问题

\* 读问题:

\* 脏读:

\* 不可重复读:

\* 虚读:

\* 写问题:

\* 丢失更新:

4. 如何解决安全性问题

\* 读问题解决，设置数据库隔离级别

\* 写问题解决可以使用 悲观锁和乐观锁的方式解决

## Spring框架的事务管理相关的类和API

1. PlatformTransactionManager接口 -- 平台事务管理器.(真正管理事务的类)。该接口有具体的实现类，根据不同的持久层框架，需要选择不同的实现类！

2. TransactionDefinition接口 -- 事务定义信息.(事务的隔离级别,传播行为,超时,只读)

3. TransactionStatus接口 -- 事务的状态

4. 总结：上述对象之间的关系：平台事务管理器真正管理事务对象.根据事务定义的信息TransactionDefinition 进行事务管理，在管理事务中产生一些状态.将状态记录到TransactionStatus中

**5. PlatformTransactionManager接口中实现类和常用的方法**

1. 接口的实现类

\* 如果使用的Spring的JDBC模板或者MyBatis框架，需要选择DataSourceTransactionManager实现类

\* 如果使用的是Hibernate的框架，需要选择HibernateTransactionManager实现类

2. 该接口的常用方法

\* void commit(TransactionStatus status)

\* TransactionStatus getTransaction(TransactionDefinition definition)

\* void rollback(TransactionStatus status)

6. TransactionDefinition

1. 事务隔离级别的常量

\* static int ISOLATION\_DEFAULT -- 采用数据库的默认隔离级别

\* static int ISOLATION\_READ\_UNCOMMITTED

\* static int ISOLATION\_READ\_COMMITTED

\* static int ISOLATION\_REPEATABLE\_READ

\* static int ISOLATION\_SERIALIZABLE

2. 事务的传播行为常量（不用设置，使用默认值）

\* 先解释什么是事务的传播行为：解决的是业务层之间的方法调用！！

\* PROPAGATION\_REQUIRED（默认值） -- A中有事务,使用A中的事务.如果没有，B就会开启一个新的事务,将A包含进来.(保证A,B在同一个事务中)，默认值！！

\* PROPAGATION\_SUPPORTS -- A中有事务,使用A中的事务.如果A中没有事务.那么B也不使用事务.

\* PROPAGATION\_MANDATORY -- A中有事务,使用A中的事务.如果A没有事务.抛出异常.

\* PROPAGATION\_REQUIRES\_NEW（记）-- A中有事务,将A中的事务挂起.B创建一个新的事务.(保证A,B没有在一个事务中)

\* PROPAGATION\_NOT\_SUPPORTED -- A中有事务,将A中的事务挂起.

\* PROPAGATION\_NEVER -- A中有事务,抛出异常.

\* PROPAGATION\_NESTED（记） -- 嵌套事务.当A执行之后,就会在这个位置设置一个保存点.如果B没有问题.执行通过.如果B出现异常,运行客户根据需求回滚(选择回滚到保存点或者是最初始状态)

## 搭建事务管理转账案例的环境（强调：简化开发，以后DAO可以继承JdbcDaoSupport类）

1. 步骤一：创建WEB工程，引入需要的jar包

\* IOC的6个包

\* AOP的4个包

\* C3P0的1个包

\* MySQL的驱动包

\* JDBC目标2个包

\* 整合JUnit测试包

2. 步骤二：引入配置文件

\* 引入配置文件

\* 引入log4j.properties

\* 引入applicationContext.xml

<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">

<property name="driverClass" value="com.mysql.jdbc.Driver"/>

<property name="jdbcUrl" value="jdbc:mysql:///spring\_day03"/>

<property name="user" value="root"/>

<property name="password" value="root"/>

</bean>

3. 步骤三：创建对应的包结构和类

\* com.qf.demo1

\* AccountService

\* AccountServlceImpl

\* AccountDao

\* AccountDaoImpl

4. 步骤四:引入Spring的配置文件,将类配置到Spring中

<bean id="accountService" class="com.qf.demo1.AccountServiceImpl">

</bean>

<bean id="accountDao" class="com.qf.demo1.AccountDaoImpl">

</bean>

5. 步骤五：在业务层注入DAO ,在DAO中注入JDBC模板（强调：简化开发，以后DAO可以继承JdbcDaoSupport类）

<bean id="accountService" class="com.qf.demo1.AccountServiceImpl">

<property name="accountDao" ref="accountDao"/>

</bean>

<bean id="accountDao" class="com.qf.demo1.AccountDaoImpl">

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

</bean>

6. 步骤六：编写DAO和Service中的方法

public class AccountDaoImpl extends JdbcDaoSupport implements AccountDao {

public void outMoney(String out, double money) {

this.getJdbcTemplate().update("update t\_account set money = money = ? where name = ?", money,out);

}

public void inMoney(String in, double money) {

this.getJdbcTemplate().update("update t\_account set money = money + ? where name = ?", money,in);

}

}

7. 步骤七：编写测试程序.

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration("classpath:applicationContext.xml")

public class Demo1 {

@Resource(name="accountService")

private AccountService accountService;

@Test

public void run1(){

accountService.pay("冠希", "美美", 1000);

}

}

## Spring框架的事务管理的分类

1. Spring的事务管理的分类

1. Spring的编程式事务管理（不推荐使用）

通过手动编写代码的方式完成事务的管理（不推荐）

2. Spring的声明式事务管理（底层采用AOP的技术）

通过一段配置的方式完成事务的管理（重点掌握注解的方式）

## Spring框架的事务管理之编程式的事务管理（了解）

1. 说明：Spring为了简化事务管理的代码:提供了模板类 TransactionTemplate，所以手动编程的方式来管理事务，只需要使用该模板类即可！！

2. 手动编程方式的具体步骤如下：

1. 步骤一:配置一个事务管理器，Spring使用PlatformTransactionManager接口来管理事务，所以咱们需要使用到他的实现类！！

<!-- 配置事务管理器 -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

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

</bean>

2. 步骤二:配置事务管理的模板

<!-- 配置事务管理的模板 -->

<bean id="transactionTemplate" class="org.springframework.transaction.support.TransactionTemplate">

<property name="transactionManager" ref="transactionManager"/>

</bean>

3. 步骤三:在需要进行事务管理的类中,注入事务管理的模板.

<bean id="accountService" class="com.qf.demo1.AccountServiceImpl">

<property name="accountDao" ref="accountDao"/>

<property name="transactionTemplate" ref="transactionTemplate"/>

</bean>

4. 步骤四:在业务层使用模板管理事务:

// 注入事务模板对象

private TransactionTemplate transactionTemplate;

public void setTransactionTemplate(TransactionTemplate transactionTemplate) {

this.transactionTemplate = transactionTemplate;

}

public void pay(final String out, final String in, final double money) {

transactionTemplate.execute(new TransactionCallbackWithoutResult() {

protected void doInTransactionWithoutResult(TransactionStatus status) {

// 扣钱

accountDao.outMoney(out, money);

int a = 10/0;

// 加钱

accountDao.inMoney(in, money);

}

});

}

## Spring框架的事务管理之声明式事务管理

即通过配置文件来完成事务管理(AOP思想)

1. 声明式事务管理又分成两种方式

基于AspectJ的XML方式（重点掌握）

基于AspectJ的注解方式（重点掌握）

### Spring框架的事务管理之基于AspectJ的XML方式（重点掌握）

1. 步骤一:恢复转账开发环境

2. 步骤二:引入AOP的开发包

3. 步骤三: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>

4. 步骤四:配置事务增强

<!-- 配置事务增强 -->

<tx:advice id="txAdvice" transaction-manager="transactionManager">

<tx:attributes>

<!--

name ：绑定事务的方法名，可以使用通配符，可以配置多个。

propagation ：传播行为

isolation ：隔离级别

read-only ：是否只读

timeout ：超时信息

rollback-for：发生哪些异常回滚.

no-rollback-for：发生哪些异常不回滚.

-->

<!-- 哪些方法加事务 -->

<tx:method name="pay" propagation="REQUIRED"/>

</tx:attributes>

</tx:advice>

5. 步骤五:配置AOP的切面

<!-- 配置AOP切面产生代理 -->

<aop:config>

<aop:advisor advice-ref="myAdvice" pointcut="execution(\* com.qf.demo2.AccountServiceImpl.pay(..))"/>

</aop:config>

**注意：如果是自己编写的切面，使用<aop:aspect>标签，如果是系统制作的，使用<aop:advisor>标签。**

6. 步骤六:编写测试类

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration("classpath:applicationContext2.xml")

public class Demo2 {

@Resource(name="accountService")

private AccountService accountService;

@Test

public void run1(){

accountService.pay("冠希", "美美", 1000);

}

}

## Spring框架的事务管理之基于AspectJ的注解方式（重点掌握，最简单的方式）

1. 步骤一:恢复转账的开发环境

2. 步骤二: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>

3. 步骤三:开启注解事务

<!-- 开启注解事务 -->

<tx:annotation-driven transaction-manager="transactionManager"/>

4. 步骤四:在业务层上添加一个注解:@Transactional

5. 编写测试类

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration("classpath:applicationContext3.xml")

public class Demo3 {

@Resource(name="accountService")

private AccountService accountService;

@Test

public void run1(){

accountService.pay("冠希", "美美", 1000);

}

}