Spring – day03

学习目标

* 能够编写两种动态代理代码
* 能够描述aop概念及应用场景
* 能够描述aop相关术语
* 能够使用xml的方式和注解的方式配置aop

学习指南

动态代理（重点）

AOP的概念

spring中的AOP相关术语

spring中基于XML和注解的AOP配置（重点）

# 第一堂课

**本节知识点**

1.1 今日课程内容介绍

1.2 案例中添加转账方法并演示事务问题

1.3 分析事务的问题并编写ConnectionUtils

1.4 编写事务管理工具类并分析连接和线程解绑

**本节目标**

编写转账的代码

能够编写ConnnectionUtils工具类

能够编写事务管理工具类和线程解绑方法

**视频时间：**

36分49秒

## 今日课程内容介绍

### 知识概述

清楚今日讲解内容

### 视频详情



### 总结与补充

1、完善我们的account案例

2、分析案例中问题

3、回顾之前讲过的一个技术：动态代理

4、动态代理另一种实现方式

5、解决案例中的问题

6、AOP的概念

7、spring中的AOP相关术语

8、spring中基于XML和注解的AOP配置

### 课堂提问与练习

无

### 习题答案

无

## 案例中添加转账方法并演示事务问题

### 知识概述

搭建工程，演示转账案例中的问题

### 视频详情



### 总结与补充

新建项目day03\_eesy\_01account导入第二天工程day02\_eesy\_02account\_xmlioc中的代码

修改IAccountService及AccountServiceImpl

IAccountService.java新增方法

|  |
| --- |
| /\*\*  \* 转账  \* @param sourceName 转出账户名称  \* @param targetName 转入账户名称  \* @param money 转账金额  \*/  void transfer(String sourceName,String targetName,Float money); |

AccountServiceImpl.java新增方法

|  |
| --- |
| @Override  public void transfer(String sourceName, String targetName, Float money) {  System.out.println("transfer....");  //2.1根据名称查询转出账户  Account source = accountDao.findAccountByName(sourceName);  //2.2根据名称查询转入账户  Account target = accountDao.findAccountByName(targetName);  //2.3转出账户减钱  source.setMoney(source.getMoney()-money);  //2.4转入账户加钱  target.setMoney(target.getMoney()+money);  //2.5更新转出账户  accountDao.updateAccount(source);  // int i=1/0;  //2.6更新转入账户  accountDao.updateAccount(target);  } |

修改IAccountDao及AccountDaoImpl中的代码

IAccountDao.java新增方法

|  |
| --- |
| /\*\*  \* 根据名称查询账户  \* @param accountName  \* @return 如果有唯一的一个结果就返回，如果没有结果就返回null  \* 如果结果集超过一个就抛异常  \*/  Account findAccountByName(String accountName); |

AccountDaoImpl新增方法

|  |
| --- |
| @Override  public Account findAccountByName(String accountName) {  try{  List<Account> accounts = runner.query(connectionUtils.getThreadConnection(),"select \* from account where name = ? ",new BeanListHandler<Account>(Account.class),accountName);  if(accounts == null || accounts.size() == 0){  return null;  }  if(accounts.size() > 1){  throw new RuntimeException("结果集不唯一，数据有问题");  }  return accounts.get(0);  }catch (Exception e) {  throw new RuntimeException(e);  }  } |

测试代码：

|  |
| --- |
| /\*\*  \* 使用Junit单元测试：测试我们的配置  \*/  @RunWith(SpringJUnit4ClassRunner.class)  @ContextConfiguration(locations = "classpath:bean.xml")  public class AccountServiceTest {  @Autowired  @Qualifier("proxyAccountService")  private IAccountService as;  @Test  public void testTransfer(){  as.transfer("aaa","bbb",100f);  }  } |

### 课堂提问与练习

无

### 习题答案

无

## 分析事务的问题并编写ConnectionUtils

### 知识概述

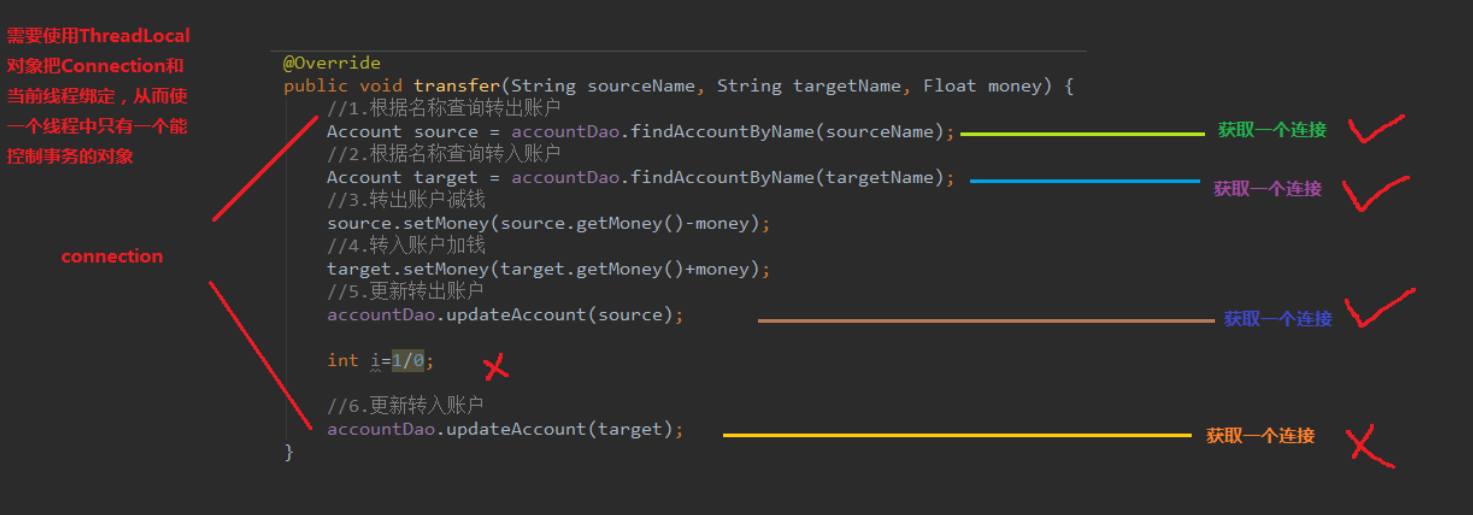
分析当前代码事务的问题，编写数据连接工具类

### 视频详情



### 总结与补充

转账失败思路分析



解决：让连接绑定到ThreadLocal中

工具类ConnectionUtils.java

|  |
| --- |
| public class ConnectionUtils {  private ThreadLocal<Connection> tl = new ThreadLocal<Connection>();  private DataSource dataSource;  public void setDataSource(DataSource dataSource) {  this.dataSource = dataSource;  }  /\*\*  \* 获取当前线程上的连接  \* @return  \*/  public Connection getThreadConnection() {  try{  //1.先从ThreadLocal上获取  Connection conn = tl.get();  //2.判断当前线程上是否有连接  if (conn == null) {  //3.从数据源中获取一个连接，并且存入ThreadLocal中  conn = dataSource.getConnection();  tl.set(conn);  }  //4.返回当前线程上的连接  return conn;  }catch (Exception e){  throw new RuntimeException(e);  }  }  } |

### 课堂提问与练习

无

### 习题答案

无

## 编写事务管理工具类并分析连接和线程解绑

### 知识概述

分析事务管理工具类

### 视频详情



### 总结与补充

编写TransactionManager事务管理工具类

TransactionManager.java

|  |
| --- |
| /\*\*  \* 和事务管理相关的工具类，它包含了，开启事务，提交事务，回滚事务和释放连接  \*/  public class TransactionManager {  private ConnectionUtils connectionUtils;  public void setConnectionUtils(ConnectionUtils connectionUtils) {  this.connectionUtils = connectionUtils;  }  /\*\*  \* 开启事务  \*/  public void beginTransaction(){  try {  connectionUtils.getThreadConnection().setAutoCommit(false);  }catch (Exception e){  e.printStackTrace();  }  }  /\*\*  \* 提交事务  \*/  public void commit(){  try {  connectionUtils.getThreadConnection().commit();  }catch (Exception e){  e.printStackTrace();  }  }  /\*\*  \* 回滚事务  \*/  public void rollback(){  try {  connectionUtils.getThreadConnection().rollback();  }catch (Exception e){  e.printStackTrace();  }  }  /\*\*  \* 释放连接  \*/  public void release(){  try {  connectionUtils.getThreadConnection().close();//还回连接池中  connectionUtils.removeConnection();  }catch (Exception e){  e.printStackTrace();  }  }  } |

在ConnectionUtils类中添加解除与线程绑定连接的方法

|  |
| --- |
| /\*\*  \* 把连接和线程解绑  \*/  public void removeConnection(){  tl.remove();  } |

### 课堂提问与练习

无

### 习题答案

无

# 第二堂课

**本节知识点**

2.1 编写业务层和持久层事务控制代码并配置spring的ioc

2.2 测试转账并分析案例中的问题

2.3 代理的分析

**本节目标**

能够理解修改后的代码作用

测试转账并分析案例中的问题

能够理解动态代理机制

**视频时间：**

27分钟29秒

## 编写业务层和持久层事务控制代码并配置spring的ioc

### 知识概述

编写service和dao的代码

### 视频详情



### 总结与补充

在service层统统都手动开启提交事务

|  |
| --- |
| public class AccountServiceImpl\_OLD implements IAccountService{  private IAccountDao accountDao;  private TransactionManager txManager;  public void setTxManager(TransactionManager txManager) {  this.txManager = txManager;  }  public void setAccountDao(IAccountDao accountDao) {  this.accountDao = accountDao;  }  @Override  public List<Account> findAllAccount() {  try {  //1.开启事务  txManager.beginTransaction();  //2.执行操作  List<Account> accounts = accountDao.findAllAccount();  //3.提交事务  txManager.commit();  //4.返回结果  return accounts;  }catch (Exception e){  //5.回滚操作  txManager.rollback();  throw new RuntimeException(e);  }finally {  //6.释放连接  txManager.release();  }  }  @Override  public Account findAccountById(Integer accountId) {  try {  //1.开启事务  txManager.beginTransaction();  //2.执行操作  Account account = accountDao.findAccountById(accountId);  //3.提交事务  txManager.commit();  //4.返回结果  return account;  }catch (Exception e){  //5.回滚操作  txManager.rollback();  throw new RuntimeException(e);  }finally {  //6.释放连接  txManager.release();  }  }  @Override  public void saveAccount(Account account) {  try {  //1.开启事务  txManager.beginTransaction();  //2.执行操作  accountDao.saveAccount(account);  //3.提交事务  txManager.commit();  }catch (Exception e){  //4.回滚操作  txManager.rollback();  }finally {  //5.释放连接  txManager.release();  }  }  @Override  public void updateAccount(Account account) {  try {  //1.开启事务  txManager.beginTransaction();  //2.执行操作  accountDao.updateAccount(account);  //3.提交事务  txManager.commit();  }catch (Exception e){  //4.回滚操作  txManager.rollback();  }finally {  //5.释放连接  txManager.release();  }  }  @Override  public void deleteAccount(Integer acccountId) {  try {  //1.开启事务  txManager.beginTransaction();  //2.执行操作  accountDao.deleteAccount(acccountId);  //3.提交事务  txManager.commit();  }catch (Exception e){  //4.回滚操作  txManager.rollback();  }finally {  //5.释放连接  txManager.release();  }  }  @Override  public void transfer(String sourceName, String targetName, Float money) {  try {  //1.开启事务  txManager.beginTransaction();  //2.执行操作  //2.1根据名称查询转出账户  Account source = accountDao.findAccountByName(sourceName);  //2.2根据名称查询转入账户  Account target = accountDao.findAccountByName(targetName);  //2.3转出账户减钱  source.setMoney(source.getMoney()-money);  //2.4转入账户加钱  target.setMoney(target.getMoney()+money);  //2.5更新转出账户  accountDao.updateAccount(source);  int i=1/0;  //2.6更新转入账户  accountDao.updateAccount(target);  //3.提交事务  txManager.commit();  }catch (Exception e){  //4.回滚操作  txManager.rollback();  e.printStackTrace();  }finally {  //5.释放连接  txManager.release();  }  }  } |

在dao层的实现类中加入连接

|  |
| --- |
| /\*\*  \* 账户的持久层实现类  \*/  public class AccountDaoImpl implements IAccountDao {  private QueryRunner runner;  private ConnectionUtils connectionUtils;  public void setRunner(QueryRunner runner) {  this.runner = runner;  }  public void setConnectionUtils(ConnectionUtils connectionUtils) {  this.connectionUtils = connectionUtils;  }  @Override  public List<Account> findAllAccount() {  try{  return runner.query(connectionUtils.getThreadConnection(),"select \* from account",new BeanListHandler<Account>(Account.class));  }catch (Exception e) {  throw new RuntimeException(e);  }  }  @Override  public Account findAccountById(Integer accountId) {  try{  return runner.query(connectionUtils.getThreadConnection(),"select \* from account where id = ? ",new BeanHandler<Account>(Account.class),accountId);  }catch (Exception e) {  throw new RuntimeException(e);  }  }  @Override  public void saveAccount(Account account) {  try{  runner.update(connectionUtils.getThreadConnection(),"insert into account(name,money)values(?,?)",account.getName(),account.getMoney());  }catch (Exception e) {  throw new RuntimeException(e);  }  }  @Override  public void updateAccount(Account account) {  try{  runner.update(connectionUtils.getThreadConnection(),"update account set name=?,money=? where id=?",account.getName(),account.getMoney(),account.getId());  }catch (Exception e) {  throw new RuntimeException(e);  }  }  @Override  public void deleteAccount(Integer accountId) {  try{  runner.update(connectionUtils.getThreadConnection(),"delete from account where id=?",accountId);  }catch (Exception e) {  throw new RuntimeException(e);  }  }  @Override  public Account findAccountByName(String accountName) {  try{  List<Account> accounts = runner.query(connectionUtils.getThreadConnection(),"select \* from account where name = ? ",new BeanListHandler<Account>(Account.class),accountName);  if(accounts == null || accounts.size() == 0){  return null;  }  if(accounts.size() > 1){  throw new RuntimeException("结果集不唯一，数据有问题");  }  return accounts.get(0);  }catch (Exception e) {  throw new RuntimeException(e);  }  }  } |

修改bean.xml文件，Bean.xml文件修改，不在配置文件中注入数据源，这一步已经在dao层做完了

|  |
| --- |
| <!-- 配置Service -->  <bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl">  <!-- 注入dao -->  <property name="accountDao" ref="accountDao"></property>  <property name="txManager" ref="txManager"></property>  </bean>  <!--配置Dao对象-->  <bean id="accountDao" class="com.itheima.dao.impl.AccountDaoImpl">  <!-- 注入QueryRunner -->  <property name="runner" ref="runner"></property>  <!-- 注入ConnectionUtils -->  <property name="connectionUtils" ref="connectionUtils"></property>  </bean>  <!--配置QueryRunner-->  <bean id="runner" class="org.apache.commons.dbutils.QueryRunner" scope="prototype"></bean>  <!-- 配置数据源 -->  <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  <!--连接数据库的必备信息-->  <property name="driverClass" value="com.mysql.jdbc.Driver"></property>  <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/eesy"></property>  <property name="user" value="root"></property>  <property name="password" value="1234"></property>  </bean>  <!-- 配置Connection的工具类 ConnectionUtils -->  <bean id="connectionUtils" class="com.itheima.utils.ConnectionUtils">  <!-- 注入数据源-->  <property name="dataSource" ref="dataSource"></property>  </bean>  <!-- 配置事务管理器-->  <bean id="txManager" class="com.itheima.utils.TransactionManager">  <!-- 注入ConnectionUtils -->  <property name="connectionUtils" ref="connectionUtils"></property>  </bean> |

### 课堂提问与练习

无

### 习题答案

无

## 测试转账并分析案例中的问题

### 知识概述

分析当前案例中的问题

### 视频详情



### 总结与补充

通过对业务层改造，已经可以实现事务控制了，但是由于我们添加了事务控制，也产生了一

个新的问题：

业务层方法变得臃肿了，里面充斥着很多重复代码。并且业务层方法和事务控制方法耦合了。

试想一下，如果我们此时提交，回滚，释放资源中任何一个方法名变更，都需要修改业务层的代码，况且这还

只是一个业务层实现类，而实际的项目中这种业务层实现类可能有十几个甚至几十个。

思考：

这个问题能不能解决呢？

答案是肯定的，使用下一小节中提到的技术

### 课堂提问与练习

无

## 代理的分析

### 知识概述

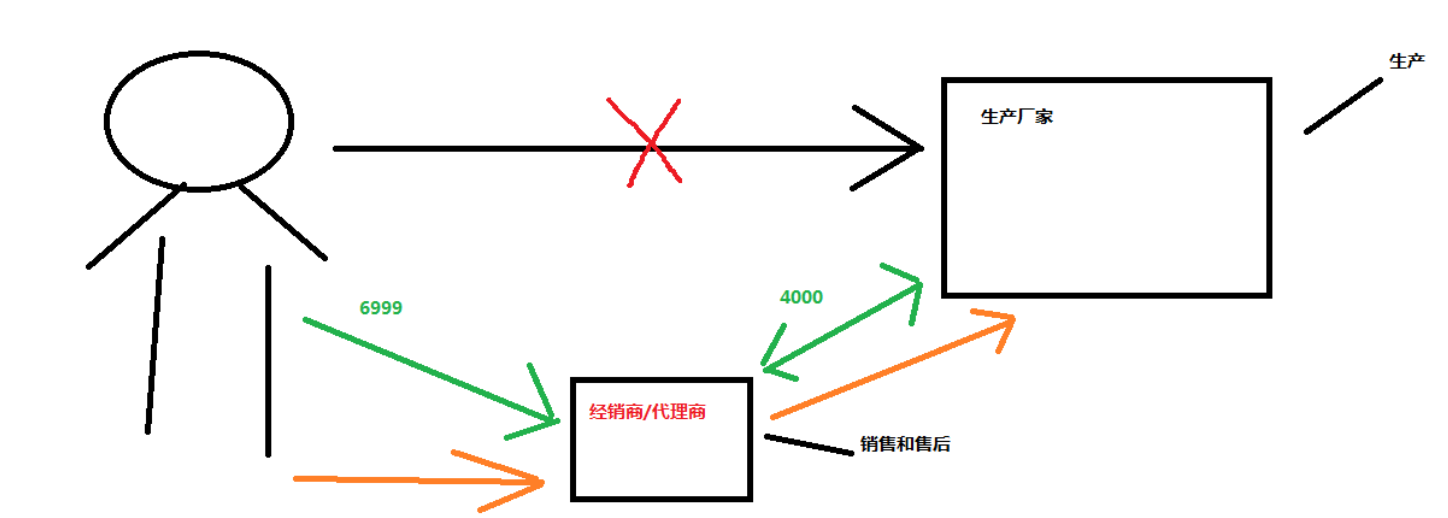
画图讲解动态代理机制

### 视频详情



### 总结与补充

看图分析动态代理



### 课堂提问与练习

无

### 习题答案

无

# 第三堂课

**本节知识点**

3.1 基于接口的动态代理回顾

3.2 基于子类的动态代理

**本节目标**

能够编写jdk动态代理

能够编写cglib实现的动态代理的代码

**视频时间：**

30分47秒

## 基于接口的动态代理回顾

### 知识概述

基于jdk的动态代理分析，并编写代码

### 视频详情



### 总结与补充

创建一个接口

|  |
| --- |
| /\*\*  \* 对生产厂家要求的接口  \*/  public interface IProducer {  /\*\*  \* 销售  \* @param money  \*/  public void saleProduct(float money);  /\*\*  \* 售后  \* @param money  \*/  public void afterService(float money);  } |

实现类

|  |
| --- |
| /\*\*  \* 一个生产者  \*/  public class Producer implements IProducer{  /\*\*  \* 销售  \* @param money  \*/  public void saleProduct(float money){  System.out.println("销售产品，并拿到钱："+money);  }  /\*\*  \* 售后  \* @param money  \*/  public void afterService(float money){  System.out.println("提供售后服务，并拿到钱："+money);  }  } |

测试类

|  |
| --- |
| /\*\*  \* 模拟一个消费者  \*/  public class Client {  public static void main(String[] args) {  final Producer producer = new Producer();  /\*\*  \* 动态代理：  \* 特点：字节码随用随创建，随用随加载  \* 作用：不修改源码的基础上对方法增强  \* 分类：  \* 基于接口的动态代理  \* 基于子类的动态代理  \* 基于接口的动态代理：  \* 涉及的类：Proxy  \* 提供者：JDK官方  \* 如何创建代理对象：  \* 使用Proxy类中的newProxyInstance方法  \* 创建代理对象的要求：  \* 被代理类最少实现一个接口，如果没有则不能使用  \* newProxyInstance方法的参数：  \* ClassLoader：类加载器  \* 它是用于加载代理对象字节码的。和被代理对象使用相同的类加载器。固定写法。  \* Class[]：字节码数组  \* 它是用于让代理对象和被代理对象有相同方法。固定写法。  \* InvocationHandler：用于提供增强的代码  \* 它是让我们写如何代理。我们一般都是些一个该接口的实现类，通常情况下都是匿名内部类，但不是必须的。  \* 此接口的实现类都是谁用谁写。  \*/  IProducer proxyProducer = (IProducer) Proxy.newProxyInstance(producer.getClass().getClassLoader(),  producer.getClass().getInterfaces(),  new InvocationHandler() {  /\*\*  \* 作用：执行被代理对象的任何接口方法都会经过该方法  \* 方法参数的含义  \* @param proxy 代理对象的引用  \* @param method 当前执行的方法  \* @param args 当前执行方法所需的参数  \* @return 和被代理对象方法有相同的返回值  \* @throws Throwable  \*/  @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  //提供增强的代码  Object returnValue = null;  //1.获取方法执行的参数  Float money = (Float)args[0];  //2.判断当前方法是不是销售  if("saleProduct".equals(method.getName())) {  returnValue = method.invoke(producer, money\*0.8f);  }  return returnValue;  }  });  proxyProducer.saleProduct(10000f);  }  } |

### 课堂提问与练习

Jdk动态里有什么局限性？

### 习题答案

Jdk实现的动态代理，只能给有接口的类做代理，没有接口的做不代理

## 基于子类的动态代理

### 知识概述

基于cglib动态代理的代码编写

### 视频详情



### 总结与补充

创建一个类，不用实现任何接口，也可以实现接口，cglib动态代理，有没有接口都可以做代理。

|  |
| --- |
| \*\*  \* 一个生产者  \*/  public class Producer {  /\*\*  \* 销售  \* @param money  \*/  public void saleProduct(float money){  System.out.println("销售产品，并拿到钱："+money);  }  /\*\*  \* 售后  \* @param money  \*/  public void afterService(float money){  System.out.println("提供售后服务，并拿到钱："+money);  }  } |

测试类：

|  |
| --- |
| /\*\*  \* 模拟一个消费者  \*/  public class Client {  public static void main(String[] args) {  final Producer producer = new Producer();  /\*\*  \* 动态代理：  \* 特点：字节码随用随创建，随用随加载  \* 作用：不修改源码的基础上对方法增强  \* 分类：  \* 基于接口的动态代理  \* 基于子类的动态代理  \* 基于子类的动态代理：  \* 涉及的类：Enhancer  \* 提供者：第三方cglib库  \* 如何创建代理对象：  \* 使用Enhancer类中的create方法  \* 创建代理对象的要求：  \* 被代理类不能是最终类  \* create方法的参数：  \* Class：字节码  \* 它是用于指定被代理对象的字节码。  \*  \* Callback：用于提供增强的代码  \* 它是让我们写如何代理。我们一般都是些一个该接口的实现类，通常情况下都是匿名内部类，但不是必须的。  \* 此接口的实现类都是谁用谁写。  \* 我们一般写的都是该接口的子接口实现类：MethodInterceptor  \*/  Producer cglibProducer = (Producer)Enhancer.create(producer.getClass(), new MethodInterceptor() {  /\*\*  \* 执行北地阿里对象的任何方法都会经过该方法  \* @param proxy  \* @param method  \* @param args  \* 以上三个参数和基于接口的动态代理中invoke方法的参数是一样的  \* @param methodProxy ：当前执行方法的代理对象  \* @return  \* @throws Throwable  \*/  @Override  public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {  //提供增强的代码  Object returnValue = null;  //1.获取方法执行的参数  Float money = (Float)args[0];  //2.判断当前方法是不是销售  if("saleProduct".equals(method.getName())) {  returnValue = method.invoke(producer, money\*0.8f);  }  return returnValue;  }  });  cglibProducer.saleProduct(12000f);  }  } |

### 课堂提问与练习

无

### 习题答案

无

# 第四堂课

**本节知识点**

4.1 使用动态代理实现事务控制

4.2 AOP的概念

4.3 spring中的aop术语和细节

**本节目标**

使用动态代理实现事务控制

能够描述AOP带来的好处

能够清晰知道aop相关术语

**视频时间：**

31分09秒

## 使用动态代理实现事务控制

### 知识概述

修改之前的代码，使用动态代理的方式进行事务控制

### 视频详情



### 总结与补充

编写

|  |
| --- |
| /\*\*  \* 用于创建Service的代理对象的工厂  \*/  public class BeanFactory {  private IAccountService accountService;  private TransactionManager txManager;  public void setTxManager(TransactionManager txManager) {  this.txManager = txManager;  }  public final void setAccountService(IAccountService accountService) {  this.accountService = accountService;  }  /\*\*  \* 获取Service代理对象  \* @return  \*/  public IAccountService getAccountService() {  return (IAccountService)Proxy.newProxyInstance(accountService.getClass().getClassLoader(),  accountService.getClass().getInterfaces(),  new InvocationHandler() {  /\*\*  \* 添加事务的支持  \*  \* @param proxy  \* @param method  \* @param args  \* @return  \* @throws Throwable  \*/  @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  if("test".equals(method.getName())){  return method.invoke(accountService,args);  }  Object rtValue = null;  try {  //1.开启事务  txManager.beginTransaction();  //2.执行操作  rtValue = method.invoke(accountService, args);  //3.提交事务  txManager.commit();  //4.返回结果  return rtValue;  } catch (Exception e) {  //5.回滚操作  txManager.rollback();  throw new RuntimeException(e);  } finally {  //6.释放连接  txManager.release();  }  }  });  }  } |

修改bean.xml文件

|  |
| --- |
| <!--配置代理的service-->  <bean id="proxyAccountService" factory-bean="beanFactory" factory-method="getAccountService"></bean>  <!--配置beanfactory-->  <bean id="beanFactory" class="com.itheima.factory.BeanFactory">  <!-- 注入service -->  <property name="accountService" ref="accountService"></property>  <!-- 注入事务管理器 -->  <property name="txManager" ref="txManager"></property>  </bean>  <!-- 配置Service -->  <bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl">  <!-- 注入dao -->  <property name="accountDao" ref="accountDao"></property>  ~~<property name="txManager" ref="txManager"></property>~~  </bean> |

### 课堂提问与练习

无

### 习题答案

无

## AOP的概念

### 知识概述

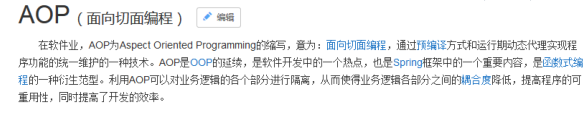
Aop概述，好处及作用

### 视频详情



### 总结与补充

AOP：全称是 Aspect Oriented Programming 即：面向切面编程。



简单的说它就是把我们程序重复的代码抽取出来，在需要执行的时候，使用动态代理的技术，在不修改源码的基础上，对我们的已有方法进行增强。

Aop的作用及好处：

作用：在程序运行期间，不修改源码对已有方法进行增强。

优势：

减少重复代码

提高开发效率

维护方便

AOP 的实现方式

使用动态代理技术

### 课堂提问与练习

简单描述一下AOP?

### 习题答案

AOP：全称是 Aspect Oriented Programming 即：面向切面编程。

简单的说它就是把我们程序重复的代码抽取出来，在需要执行的时候，使用动态代理的技术，在不修改源码的基础上，对我们的已有方法进行增强。

## spring中的aop术语和细节

### 知识概述

详细讲解aop中涉及到的相关术语

### 视频详情



### 总结与补充

**AOP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。

通知的类型：前置通知,后置通知,异常通知,最终通知,环绕通知。

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。

**学习 spring中的aop要明确的事**

a 、开发阶段（我们做的）

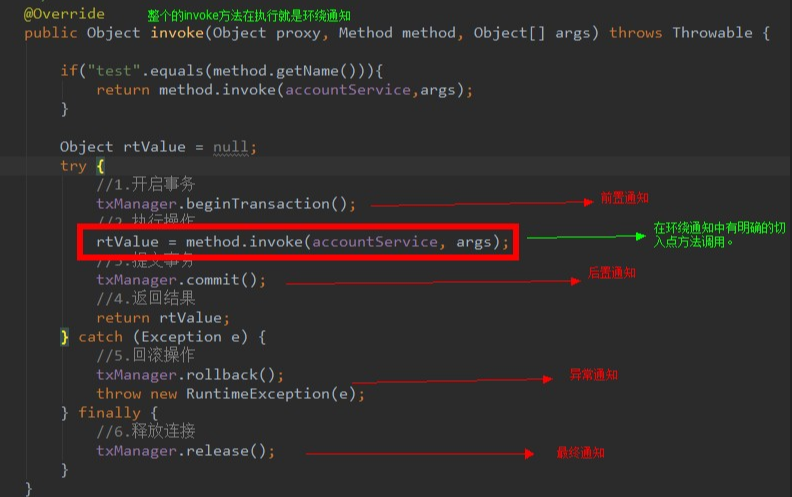
编写核心业务代码（开发主线）：大部分程序员来做，要求熟悉业务需求。

把公用代码抽取出来，制作成通知。（开发阶段最后再做）：AOP 编程人员来做。

在配置文件中，声明切入点与通知间的关系，即切面。：AOP 编程人员来做。

b 、运行阶段（Spring 框架完成的）

Spring 框架监控切入点方法的执行。一旦监控到切入点方法被运行，使用代理机制，动态创建目标对象的代理对象，根据通知类别，在代理对象的对应位置，将通知对应的功能织入，完成完整的代码逻辑运行。



### 课堂提问与练习

描述一下Pointcut，Advice，Target，Proxy，Aspect的含义

### 习题答案

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。

通知的类型：前置通知,后置通知,异常通知,最终通知,环绕通知。

Target( 目标对象):代理的目标对象。

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。

# 第五堂课

**本节知识点**

5.1 spring基于XML的AOP-编写必要的代码

5.2 spring基于XML的AOP-配置步骤

5.3 切入点表达式的写法

5.4 四种常用通知类型

**本节目标**

编写aop功能的准备代码

理解bean.xml文件中的配置含义

能够掌握切点表达式的写法

能够清楚前置，后置，异常，最终通知的配置

**视频时间：**

45分01秒

## spring基于XML的AOP-编写必要的代码

### 知识概述

编写aop功能的准备代码

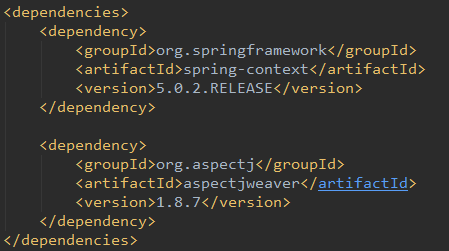
### 视频详情



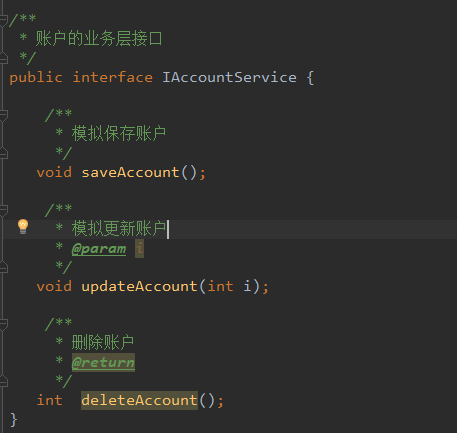
### 总结与补充

创建工程day03\_eesy\_03springAOP

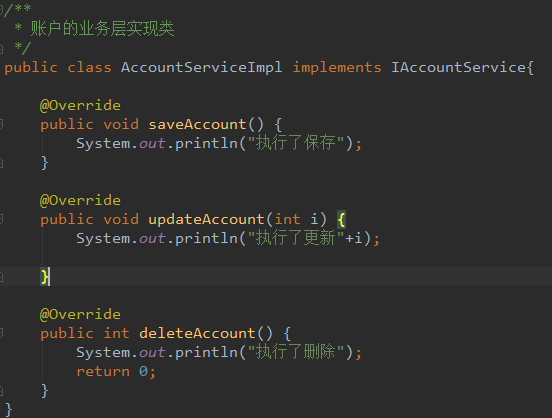
Pom.xml文件导入坐标：



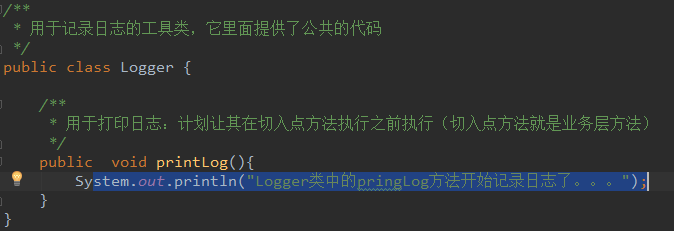
新建IAccountService接口



实现类：AccountServiceImpl



记录日志的工具类：



### 课堂提问与练习

无

### 习题答案

无

## spring基于XML的AOP-配置步骤

### 知识概述

配置bean.xml文件，进行aop相关的配置，（目标，通知，切面）

### 视频详情

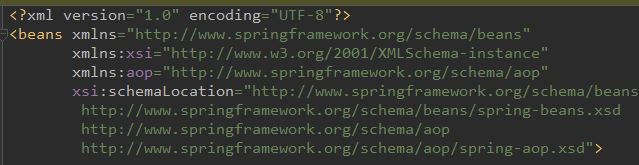


### 总结与补充

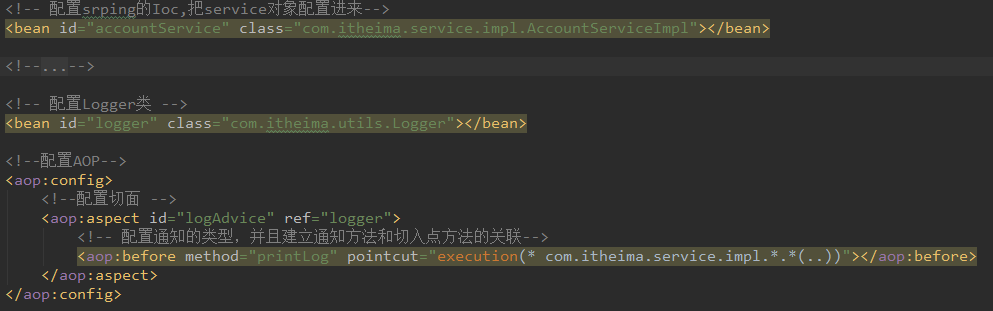
创建bean.xml文件，导入aop相关命名空间，约束路径



把aop相关的约束全部导入进来



Bean.xml文件的配置：



spring中基于XML的AOP配置步骤

1、把通知Bean也交给spring来管理

2、使用aop:config标签表明开始AOP的配置

3、使用aop:aspect标签表明配置切面

id属性：是给切面提供一个唯一标识

ref属性：是指定通知类bean的Id。

4、在aop:aspect标签的内部使用对应标签来配置通知的类型

我们现在示例是让printLog方法在切入点方法执行之前之前：所以是前置通知

aop:before：表示配置前置通知

method属性：用于指定Logger类中哪个方法是前置通知

pointcut属性：用于指定切入点表达式，该表达式的含义指的是对业务层中哪些方法增强

切入点表达式的写法：

关键字：execution(表达式)

表达式：访问修饰符 返回值 包名.包名.包名...类名.方法名(参数列表)

标准的表达式写法：

public void com.itheima.service.impl.AccountServiceImpl.saveAccount()

### 课堂提问与练习

无

### 习题答案

无

## 切入点表达式的写法

### 知识概述

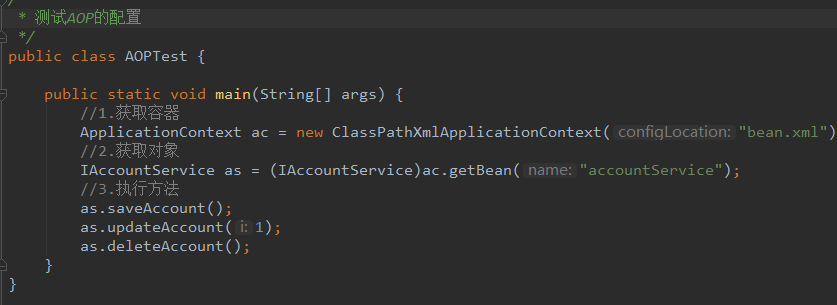
切点表达式详解

### 视频详情



### 总结与补充

创建测试类：AOPTest



**切入点表达式的写法：**

关键字：execution(表达式)

表达式：

访问修饰符 返回值 包名.包名.包名...类名.方法名(参数列表)

标准的表达式写法：

public void com.itheima.service.impl.AccountServiceImpl.saveAccount()

访问修饰符可以省略

void com.itheima.service.impl.AccountServiceImpl.saveAccount()

返回值可以使用通配符，表示任意返回值

\* com.itheima.service.impl.AccountServiceImpl.saveAccount()

包名可以使用通配符，表示任意包。但是有几级包，就需要写几个\*.

\* \*.\*.\*.\*.AccountServiceImpl.saveAccount())

包名可以使用..表示当前包及其子包

\* \*..AccountServiceImpl.saveAccount()

类名和方法名都可以使用\*来实现通配

\* \*..\*.\*()

参数列表：

可以直接写数据类型：

基本类型直接写名称 int

引用类型写包名.类名的方式 java.lang.String

可以使用通配符表示任意类型，但是必须有参数

可以使用..表示有无参数均可，有参数可以是任意类型

全通配写法：

\* \*..\*.\*(..)

实际开发中切入点表达式的通常写法：

切到业务层实现类下的所有方法

\* com.itheima.service.impl.\*.\*(..)

### 课堂提问与练习

实际开发中切入点表达式的通常怎么写?

### 习题答案

切到业务层实现类下的所有方法

\* com.itheima.service.impl.\*.\*(..)

## 四种常用通知类型

### 知识概述

前置，后置，异常，最终通知的配置

### 视频详情



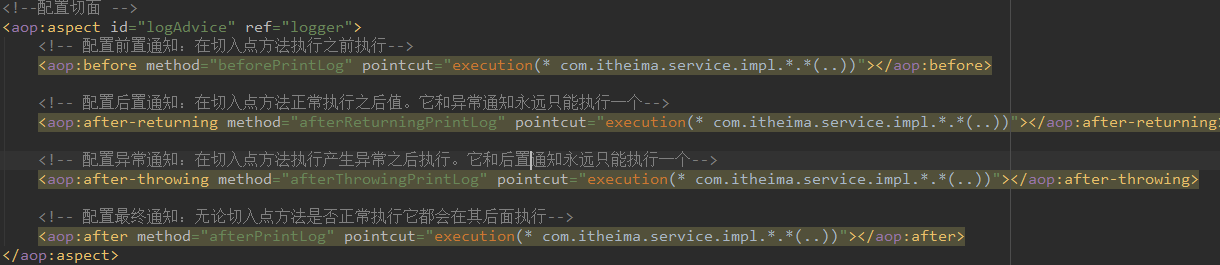
### 总结与补充

新建项目day03\_eesy\_04adviceType，导入上一个工程的代码，

修改Logger通知类



修改bean.xml文件



### 课堂提问与练习

无

### 习题答案

无

# 第六堂课

**本节知识点**

6.1 通用化切入点表达式

6.2 spring中的环绕通知

6.3 spring基于注解的AOP配置

6.4 总结和作业安排

**本节目标**

能够抽出通用的切入点表达式

能够清晰的知道环绕通知的使用

能够使用注解的方式配置通知

**视频时间：**

31分23秒

## 通用化切入点表达式

### 知识概述

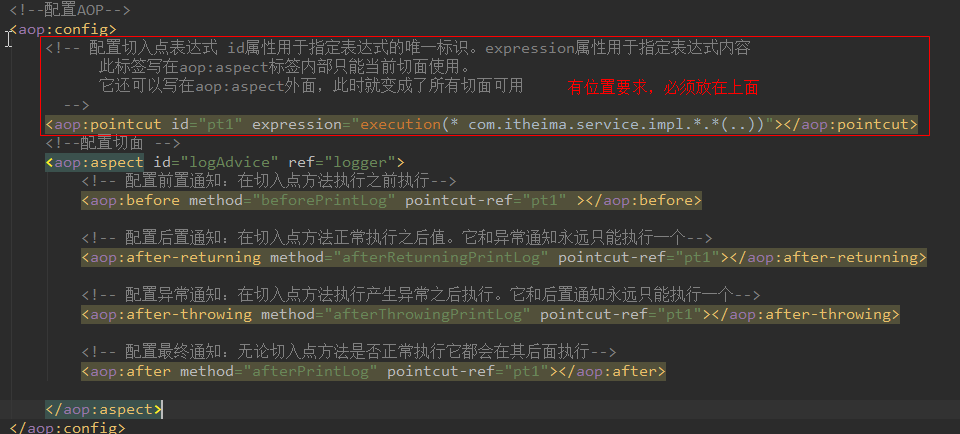
抽出通用的切入点表达式

### 视频详情



### 总结与补充

修改bean.xml文件



### 课堂提问与练习

无

### 习题答案

无

## spring中的环绕通知

### 知识概述

详细介绍了环绕通知

### 视频详情



### 总结与补充

环绕通知

问题：

当我们配置了环绕通知之后，切入点方法没有执行，而通知方法执行了。

分析：

通过对比动态代理中的环绕通知代码，发现动态代理的环绕通知有明确的切入点方法调用，而我们的代码中没有。

解决：

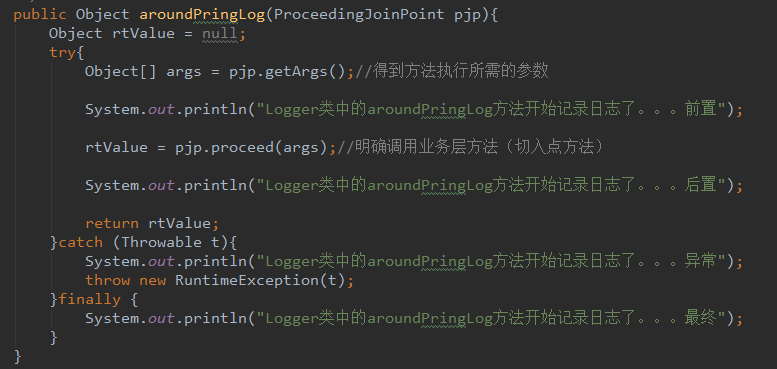
Spring框架为我们提供了一个接口：ProceedingJoinPoint。该接口有一个方法proceed()，此方法就相当于明确调用切入点方法。

该接口可以作为环绕通知的方法参数，在程序执行时，spring框架会为我们提供该接口的实现类供我们使用。

spring中的环绕通知：

它是spring框架为我们提供的一种可以在代码中手动控制增强方法何时执行的方式。

在Logger通知类中新增环绕通知方法



在bean.xml文件中配置环绕通知



### 课堂提问与练习

无

### 习题答案

无

## spring基于注解的AOP配置

### 知识概述

使用注解的方式配置前置，后置，异常，最终，环绕通知

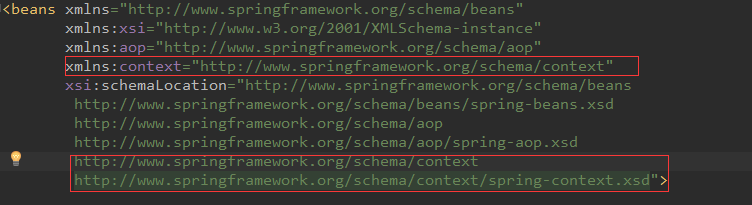
### 视频详情



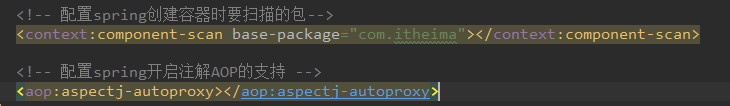
### 总结与补充

新建项目day03\_eesy\_05annotationAOP

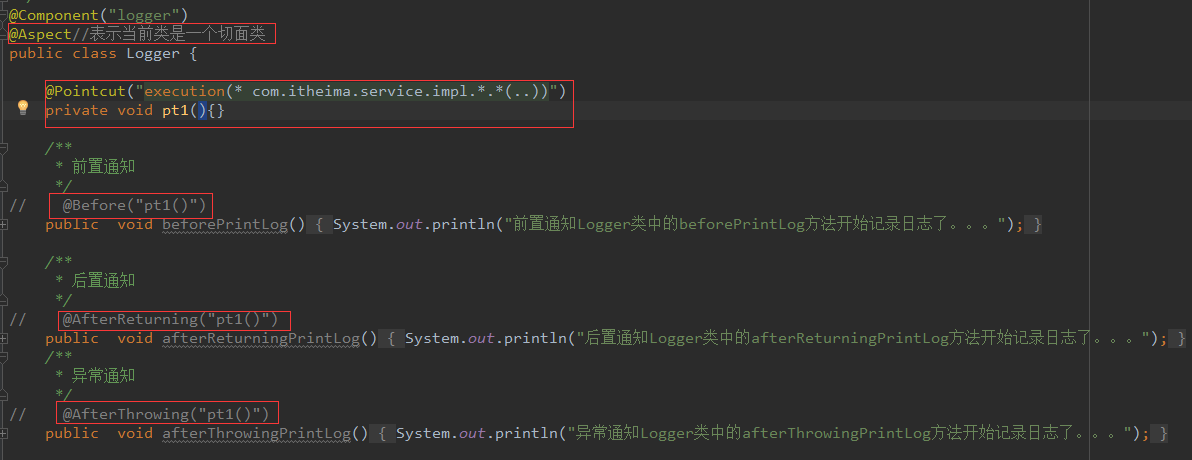
把上一个项目的代码拷贝进来，修改bean.xml文件，新增一个context的名称空间

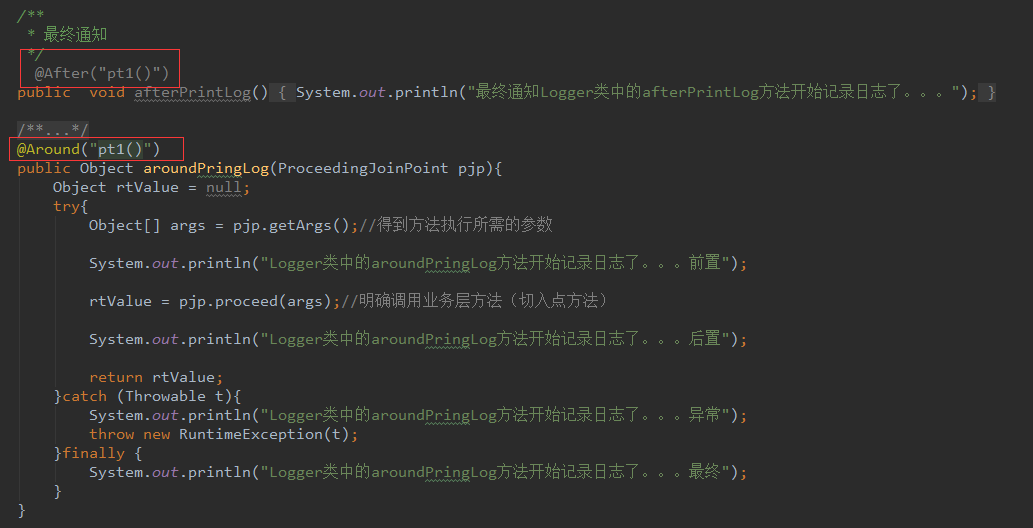


修改bean.xml文件



在Logger类中加入注解





### 课堂提问与练习

无

### 习题答案

无

## 总结和作业安排

### 知识概述

当日内容总结及作业的安排

### 视频详情



### 总结与补充

作业安排：修改day03\_eesy\_01account工程，使用aop对事务进行控制

### 课堂提问与练习

无

### 习题答案

无