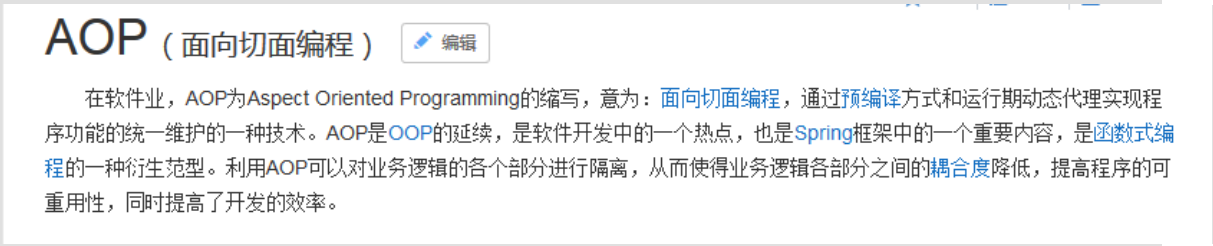
Spring 第三天

第1章 AOP 的相关概念[理解]

1.1AOP 概述

1.1.1 什么是 AOP

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。

1.1.2 AOP 的作用及优势

作用： 也是动态代理的作用

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

1.1.3 AOP 的实现方式

使用动态代理技术

1.2AOP 的具体应用

1.2.1 案例中问题

这是我们昨天课程中做的增删改查例子。下面是客户的业务层实现类。我们能看出什么问题吗？

客户的业务层实现类

```
/**
 * 账户的业务层实现类
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
public class AccountServiceImpl implements IAccountService {

    private IAccountDao accountDao;

    public void setAccountDao(IAccountDao accountDao) {
        this.accountDao = accountDao;
    }

    @Override
    public void saveAccount(Account account) throws SQLException {
        accountDao.save(account);
    }

    @Override
    public void updateAccount(Account account) throws SQLException{
        accountDao.update(account);
    }

    @Override
    public void deleteAccount(Integer accountId) throws SQLException{
        accountDao.delete(accountId);
    }

    @Override
    public Account findAccountById(Integer accountId) throws SQLException {
        return accountDao.findById(accountId);
    }

    @Override
    public List<Account> findAllAccount() throws SQLException{
        return accountDao.findAll();
    }
}
```

问题就是：

事务被自动控制了。换言之，我们使用了 connection 对象的 setAutoCommit(true)

此方式控制事务，如果我们每次都执行一条 sql 语句，没有问题，但是如果业务方法一次要执行多条 sql 语句，这种方式就无法实现功能了。

请看下面的示例：

我们在业务层中多加入一个方法。

业务层接口

```
/**
 * 转账
 * @param sourceName
 * @param targetName
 * @param money
 */
void transfer(String sourceName,String targetName,Float money);
```

业务层实现类：

```
@Override
public void transfer(String sourceName, String targetName, Float money) {
    //根据名称查询两个账户信息
    Account source = accountDao.findByName(sourceName);获取一个连接
    Account target = accountDao.findByName(targetName);获取一个连接
    //转出账户减钱，转入账户加钱
    source.setMoney(source.getMoney()-money);
    target.setMoney(target.getMoney()+money);
    //更新两个账户
    accountDao.update(source);获取一个连接
    int i=1/0; //模拟转账异常
    accountDao.update(target);获取一个连接
}
```

和数据库交互：

当我们执行时，由于执行有异常，转账失败。但是因为我们是每次执行持久层方法都是独立事务，导致无法实现事务控制（不符合事务的一致性）

解决方法：需要使用TreadLocal对象把Connection和当前线程绑定，从而使一个线程只有一个能控制事物的对象

1.2.2 问题的解决

解决办法：

让业务层来控制事务的提交和回滚。（这个我们之前已经在 web 阶段讲过了）

改造后的业务层实现类：

注：此处没有使用 spring 的 IoC.

```
/**
 * 账户的业务层实现类
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
public class AccountServiceImpl implements IAccountService {
```

```
private IAccountDao accountDao = new AccountDaoImpl();


@Override
public void saveAccount(Account account) {
    try {
        TransactionManager.beginTransaction();
        accountDao.save(account);
        TransactionManager.commit();
    } catch (Exception e) {
        TransactionManager.rollback();
        e.printStackTrace();
    }finally {
        TransactionManager.release();
    }
}

@Override
public void updateAccount(Account account) {
    try {
        TransactionManager.beginTransaction();
        accountDao.update(account);
        TransactionManager.commit();
    } catch (Exception e) {
        TransactionManager.rollback();
        e.printStackTrace();
    }finally {
        TransactionManager.release();
    }
}

@Override
public void deleteAccount(Integer accountId) {
    try {
        TransactionManager.beginTransaction();
        accountDao.delete(accountId);
        TransactionManager.commit();
    } catch (Exception e) {
        TransactionManager.rollback();
        e.printStackTrace();
    }finally {
        TransactionManager.release();
    }
}
```

```
try{
    //1.开启事务
    txManager.beginTrasaction();
    //2.执行操作
    accountDao.saveAccount(account);
    //3.提交事务
    txManager.commit();
}catch(Exception e){
    //5.回滚操作
    txManager.rollback();
}finally {
    //6.释放连接
    txManager.release();
}
```

```
@Override
public Account findAccountById(Integer accountId) {
    Account account = null;
    try {
        TransactionManager.beginTransaction();
        account = accountDao.findById(accountId);
        TransactionManager.commit();
        return account;
    } catch (Exception e) {
        TransactionManager.rollback();
        e.printStackTrace();
    }finally {
        TransactionManager.release();
    }
    return null;
}

@Override
public List<Account> findAllAccount() {
    List<Account> accounts = null;
    try {
        TransactionManager.beginTransaction();
        accounts = accountDao.findAll();
        TransactionManager.commit();
        return accounts;
    } catch (Exception e) {
        TransactionManager.rollback();
        e.printStackTrace();
    }finally {
        TransactionManager.release();
    }
    return null;
}

@Override
public void transfer(String sourceName, String targetName, Float money) {
    try {
        TransactionManager.beginTransaction();
        Account source = accountDao.findByName(sourceName);
        Account target = accountDao.findByName(targetName);
        source.setMoney(source.getMoney()-money);
        target.setMoney(target.getMoney()+money);
        accountDao.update(source);
```

```
try{
    //1.开启事务
    txManager.beginTrasaction();
    //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();
}
```

```
            int i=1/0;
            accountDao.update(target);
            TransactionManager.commit();
        } catch (Exception e) {
            TransactionManager.rollback();
            e.printStackTrace();
        }finally {
            TransactionManager.release();
        }
    }
}
```

TransactionManager 类的代码：

```
/**
 * 事务控制类
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
public class TransactionManager {

    //定义一个 DBAssit
    private static DBAssit dbAssit =  new DBAssit(C3P0Utils.getDataSource(),true);
    private ConnectionUtils connectionUtils;
    public void setConnectionUtils(ConnectionUtils connectionUtils) {
        this.connectionUtils = connectionUtils;
    }
    //开启事务
    public static void beginTransaction() {
        try {
            connectionUtils.getConnection().setAutoCommit(false);
            dbAssit.getCurrentConnection().setAutoCommit(false);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    //提交事务
    public static void commit() {
        try {
            connectionUtils.getConnection().commit();
            dbAssit.getCurrentConnection().commit();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    //回滚事务
```

都是connectionUtils.getConnection().操作

```
import javax.sql.DataSource;
import java.sql.Connection;

/**
 * 连接工具类，用于从数据源中获取一个连接，并且实现和业务的绑定
 */
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 getConnection(){
        //1.先从ThreadLocal上获取
        Connection conn=tl.get();
        try {
            //2.判断当前线程上是否有连接
            if (conn == null) {
                //3.从数据源中获取到一个连接，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);
        }
    }

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

```
        public static void rollback() {
            try {connectionUtils.getConnection().rollback();
                dbAssit.getCurrentConnection().rollback();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }

        //释放资源
        public static void release() {
            try {connectionUtils.getConnection().close();//还回连接池中
                 connectionUtils.removeConnection();
                dbAssit.releaseConnection();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

1.2.3 新的问题

上一小节的代码，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。

1.2.4 动态代理回顾

```
动态代理：
        特点：字节码随用随创建，随用随加载
        和装饰者模式的不同点：装饰者模式一开始必须写好一个类，动态代理没必要，随用随创建
        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，用于加载代理对象的字节码，和被代理对象使用相同的类加载器，固定写法：xxx.getClass().getClassLoader()
        Class[]:字节码数组，用于让代理对象和被代理对象有相同的方法，固定写法：xxx.getClass().getInterfaces()
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，用于指定被代理对象的字节码，有了被代理对象的字节码，不管是想用类加载器还是想看看有哪些内容都可以得到
        Callback:用于提供增强的字节码，让我们写如何代理，一般都是写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，此接口的实现类是谁用谁写，一般写的都是该接口的子接口实现类：MethodInterceptor
```

1.2.4.1 动态代理的特点

字节码随用随创建，随用随加载。

它与静态代理的区别也在于此。因为静态代理是字节码一上来就创建好，并完成加载。

装饰者模式就是静态代理的一种体现。

1.2.4.2 动态代理常用的有两种方式

基于接口的动态代理

提供者：JDK 官方的 Proxy 类。

要求：被代理类最少实现一个接口。

基于子类的动态代理

提供者：第三方的 CGLib，如果报 asmxxxx 异常，需要导入 asm.jar。

要求：被代理类不能用 final 修饰的类（最终类）。

1.2.4.3 使用 JDK 官方的 Proxy 类创建代理对象

此处我们使用的是一个演员的例子：

在很久以前，演员和剧组都是直接见面联系的。没有中间人环节。

而随着时间的推移，产生了一个新兴职业：经纪人（中间人），这个时候剧组再想找演员就需要通过经纪人来找了。下面我们就用代码演示出来。

```
/**
 * 一个经纪公司的要求：
 *       能做基本的表演和危险的表演
*/
public interface IActor {
    /**
     * 基本演出
     * @param money
     */
    public void basicAct(float money);
    /**
     * 危险演出
     * @param money
     */
    public void dangerAct(float money);
}

/**
 * 一个演员
 */
//实现了接口，就表示具有接口中的方法实现。即：符合经纪公司的要求
public class Actor implements IActor{

    public void basicAct(float money){
        System.out.println("拿到钱，开始基本的表演："+money);
    }

    public void dangerAct(float money){
        System.out.println("拿到钱，开始危险的表演："+money);
    }
}
```

```
public class Client {

    public static void main(String[] args) {
        //一个剧组找演员：
        final Actor actor = new Actor();//直接

        /**
         * 代理：
         *  间接。
         * 获取代理对象：
         *  要求：
         *   被代理类最少实现一个接口
         * 创建的方式
         *  Proxy.newProxyInstance(三个参数)
         * 参数含义：
         *  ClassLoader：和被代理对象使用相同的类加载器。
         *  Interfaces：和被代理对象具有相同的行为。实现相同的接口。
         *  InvocationHandler：如何代理。
         *      策略模式：使用场景是：
         *                  数据有了，目的明确。
         *                  如何达成目标，就是策略。
         *
         */
        IActor proxyActor = (IActor) Proxy.newProxyInstance(
                                    actor.getClass().getClassLoader(),
                                    actor.getClass().getInterfaces(),
                                    new InvocationHandler() {
                /**
                 * 执行被代理对象的任何方法，都会经过该方法。
                 * 此方法有拦截的功能。
                 *
                 * 参数：
                 *  proxy：代理对象的引用。不一定每次都用得到
                 *  method：当前执行的方法对象
                 *  args：执行方法所需的参数
                 * 返回值：
                 *  当前执行方法的返回值
                 */
                @Override
                public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
                    String name = method.getName();
                    Float money = (Float) args[0];
                    Object rtValue = null;
```

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   当前执行方法的参数
@return   和被代理对象方法有相同的返回值
@throws Throwable

```
                    //每个经纪公司对不同演出收费不一样，此处开始判断
                    if("basicAct".equals(name)){
                        //基本演出，没有 2000 不演
                        if(money > 2000){
                            //看上去剧组是给了 8000，实际到演员手里只有 4000
                            //这就是我们没有修改原来 basicAct 方法源码，对方法进行了增强
                            rtValue = method.invoke(actor, money/2);
                        }
                    }
                    if("dangerAct".equals(name)){
                        //危险演出,没有 5000 不演
                        if(money > 5000){
                            //看上去剧组是给了 50000，实际到演员手里只有 25000
                            //这就是我们没有修改原来 dangerAct 方法源码，对方法进行了增强
                            rtValue = method.invoke(actor, money/2);
                        }
                    }
                    return rtValue;
                }
        });
        //没有经纪公司的时候，直接找演员。
//      actor.basicAct(1000f);
//      actor.dangerAct(5000f);

        //剧组无法直接联系演员，而是由经纪公司找的演员
        proxyActor.basicAct(8000f);
        proxyActor.dangerAct(50000f);
    }
}
```

1.2.4.4 使用 CGLib 的 Enhancer 类创建代理对象

还是那个演员的例子，只不过不让他实现接口。

```
/**
 * 一个演员
*/
public class Actor{//没有实现任何接口

    public void basicAct(float money){
        System.out.println("拿到钱，开始基本的表演："+money);
    }

    public void dangerAct(float money){
        System.out.println("拿到钱，开始危险的表演："+money);
```

```
        }
    }

    public class Client {
        /**
         * 基于子类的动态代理
         *  要求：
         *      被代理对象不能是最终类
         *  用到的类：
         *      Enhancer
         *  用到的方法：
         *      create(Class, Callback)
         *  方法的参数：
         *      Class：被代理对象的字节码
         *      Callback：如何代理
         * @param args
         */
        public static void  main(String[] args) {
            final Actor actor = new Actor();

            Actor cglibActor = (Actor) Enhancer.create(actor.getClass(),
                                new MethodInterceptor() {
                /**
                 * 执行被代理对象的任何方法，都会经过该方法。在此方法内部就可以对被代理对象的任何方法进行增强。
                 *
                 * 参数：
                 *  前三个和基于接口的动态代理是一样的。
                 *  MethodProxy：当前执行方法的代理对象。
                 * 返回值：
                 *  当前执行方法的返回值
                 */
                @Override
                public Object intercept(Object proxy, Method method, Object[] args,
MethodProxy methodProxy) throws Throwable {
                    String name = method.getName();
                    Float money = (Float) args[0];
                    Object rtValue = null;
                    if("basicAct".equals(name)){
                        //基本演出
                        if(money > 2000){
                            rtValue = method.invoke(actor, money/2);
                        }
                    }
```

执行被代理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

```
                    if("dangerAct".equals(name)){
                        //危险演出
                        if(money > 5000){
                            rtValue = method.invoke(actor, money/2);
                        }
                    }
                    return rtValue;
                }
            });
            cglibActor.basicAct(10000);
            cglibActor.dangerAct(100000);
        }
    }
```

思考：

这个故事（示例）讲完之后，我们从中受到什么启发呢？它到底能应用在哪呢？

1.2.5 解决案例中的问题

```
    /**
     * 用于创建客户业务层对象工厂（当然也可以创建其他业务层对象，只不过我们此处不做那么繁琐）
     * @author 黑马程序员
     * @Company http://www.ithiema.com
     * @Version 1.0
     */
    public class BeanFactory {

        /**
         * 创建账户业务层实现类的代理对象
         * @return
         */
        public static IAccountService getAccountService() {
            //1.定义被代理对象
            final IAccountService accountService = new AccountServiceImpl();
            //2.创建代理对象
            IAccountService proxyAccountService = (IAccountService)
Proxy.newProxyInstance(accountService.getClass().getClassLoader(),
                    accountService.getClass().getInterfaces(),new
InvocationHandler() {
                        /**
                         * 执行被代理对象的任何方法，都会经过该方法。
                         * 此处添加事务控制
                         */
                        @Override
                        public Object invoke(Object proxy, Method method,
```

Target

Proxy：return返回的对象

整个invoke方法在执行的就是环绕通知

```
Object[] args) throws Throwable {
                                    Object rtValue = null;
                                    try {
                                        //开启事务
                                        TransactionManager.beginTransaction();
                                        //执行业务层方法
                                        rtValue = method.invoke(accountService, args);
                                        //提交事务
                                        TransactionManager.commit();
                                    }catch(Exception e) {
                                        //回滚事务
                                        TransactionManager.rollback();
                                        e.printStackTrace();
                                    }finally {
                                        //释放资源
                                        TransactionManager.release();
                                    }
                                    return rtValue;
                                }
                            });
            return proxyAccountService;
        }
    }
```

前置通知
在环绕通知中有明确的切入点方法调用
被代理对象
后置通知
异常通知
catch中的都算异常通知
最终通知

当我们改造完成之后，业务层用于控制事务的重复代码就都可以删掉了。

第2章 Spring 中的 AOP[掌握]

2.1Spring 中 AOP 的细节

2.1.1 说明

我们学习 spring 的 aop，就是通过配置的方式，实现上一章节的功能。配置也分为注解和xml

2.1.2 AOP 相关术语

业务层接口看到的方法都是连接点，连接业务和增强方法中的点
如何把增强的代码加到业务中来？只能依靠接口中的方法，也就是说这些方法加上业务的支持，从而这些代码形成完整的业务逻辑

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): 被增强的会成为切入点

所谓切入点是指我们要对哪些 Joinpoint 进行拦截的定义。

所有的切入点都是连接点，但是不一定所有的连接点都是切入点

动态代理中invoke方法具有拦截的功能，能拦截被代理对象中的执行的所有方法，之后干什么？
开启事务、提交事务、提交事务和释放资源
谁是通知？TranasactionManager，当拦截到之后就要进行事物的支持

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)： 原有的Service没办法实现事务的支持，于是使用动态代理技术创建一个新的对象返回一个代理对象，返回代理对象的过程中添加事务的支持，加入事务支持的过程叫织入

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)： 提供公共代码的类

是切入点和通知（引介）的结合。

哪些方法被增强过
增强过的方法就是切入点

2.1.3 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。

2.1.4 关于代理的选择

在 spring 中，框架会根据目标类是否实现了接口来决定采用哪种动态代理的方式。

2.2 基于 XML 的 AOP 配置

示例：

我们在学习 spring 的 aop 时，采用账户转账作为示例。

并且把 spring 的 ioc 也一起应用进来。

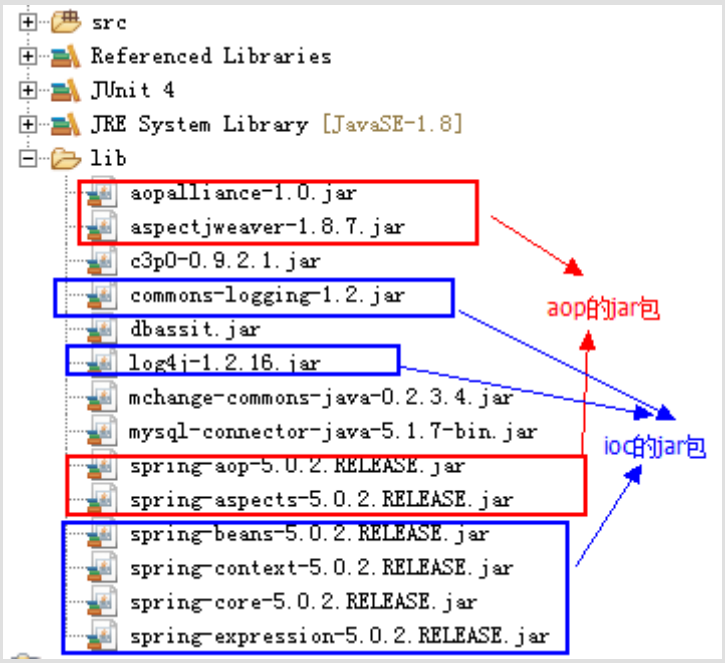
2.2.1 环境搭建

2.2.1.1 第一步：准备必要的代码

此处包含了实体类，业务层和持久层代码。我们沿用上一章节中的代码即可。

2.2.1.2 第二步：拷贝必备的 jar 包到工程的 lib 目录

此处要拷贝 spring 的 ioc 和 aop 两组 jar 包



2.2.1.3 第三步：创建 spring 的配置文件并导入约束

此处要导入 aop 的约束

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
```

2.2.1.4 第四步：配置 spring 的 ioc

```
<!-- 配置 service -->
<bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl">
    <property name="accountDao" ref="accountDao"></property>
</bean>

<!-- 配置 dao -->
<bean id="accountDao" class="com.itheima.dao.impl.AccountDaoImpl">
```

```
    <property name="dbAssit" ref="dbAssit"></property>
</bean>

<!-- 配置数据库操作对象 -->
<bean id="dbAssit" class="com.itheima.dbassit.DBAssit">
    <property name="dataSource" ref="dataSource"></property>
    <!-- 指定 connection 和线程绑定 -->
    <property name="useCurrentConnection" value="true"></property>
</bean>

<!-- 配置数据源 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
    <property name="jdbcUrl" value="jdbc:mysql:///spring_day02"></property>
    <property name="user" value="root"></property>
    <property name="password" value="1234"></property>
</bean>
```

2.2.1.5 第五步：抽取公共代码制作成通知

```
/**
 * 事务控制类
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
public class TransactionManager {

    //定义一个 DBAssit
    private DBAssit dbAssit ;

    public void setDbAssit(DBAssit dbAssit) {
        this.dbAssit = dbAssit;
    }

    //开启事务
    public void beginTransaction() {
        try {
            dbAssit.getCurrentConnection().setAutoCommit(false);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
```

```
        //提交事务
        public void commit() {
            try {
                dbAssit.getCurrentConnection().commit();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }

        //回滚事务
        public void rollback() {
            try {
                dbAssit.getCurrentConnection().rollback();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }

        //释放资源
        public void release() {
            try {
                dbAssit.releaseConnection();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

2.2.2 配置步骤

2.2.2.1 第一步：把通知类用 bean 标签配置起来

```
    <!-- 配置通知 -->
    <bean id="txManager" class="com.itheima.utils.TransactionManager">
        <property name="dbAssit" ref="dbAssit"></property>
    </bean>
```

2.2.2.2 第二步：使用 aop:config 声明 aop 配置

```
    aop:config:
        作用：用于声明开始 aop 的配置
    <aop:config>
```

```
spring中基于xml的AOP配置步骤
    1.把通知的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：给切面提供一个唯一标识
            ref属性；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
            现在的示例是让printLog方法在切入点方法执行之前执行，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，该表达式的含义是对业务层中哪些方法进行增强
```

```
    <!-- 配置的代码都写在此处 -->
</aop:config>
```

2.2.2.3 第三步：使用 aop:aspect 配置切面

```
aop:aspect:
    作用：
        用于配置切面。
    属性：
        id：给切面提供一个唯一标识。
        ref：引用配置好的通知类 bean 的 id。
<aop:aspect id="txAdvice" ref="txManager">
        <!--配置通知的类型要写在此处-->
</aop:aspect>
```

2.2.2.4 第四步：使用 aop:pointcut 配置切入点表达式

```
aop:pointcut:
    作用：
        用于配置切入点表达式。就是指定对哪些类的哪些方法进行增强。
    属性：
        expression：用于定义切入点表达式。
        id：用于给切入点表达式提供一个唯一标识
<aop:pointcut expression="execution(
    public void com.itheima.service.impl.AccountServiceImpl.transfer(
            java.lang.String, java.lang.String, java.lang.Float)
)" id="pt1"/>
```

2.2.2.5 第五步：使用 aop:xxx 配置对应的通知类型

配置前置通知:在切入点方法执行之前执行
配置后置通知:在切入点方法执行之后执行，它会和异常通知永远只能执行一个
配置异常通知:在切入点方法执行产生异常之后执行，它会和后置通知永远只能执行一个
配置最终通知:无论切入点方法是否正常执行，它都会在其后面执行

```
aop:before
    作用：
        用于配置前置通知。指定增强的方法在切入点方法之前执行
    属性：
        method:用于指定通知类中的增强方法名称
        ponitcut-ref：用于指定切入点的表达式的引用
        poinitcut：用于指定切入点表达式
    执行时间点：
        切入点方法执行之前执行
<aop:before method="beginTransaction" pointcut-ref="pt1"/>

aop:after-returning
```

作用：

用于配置后置通知

属性：

method： 指定通知中方法的名称。

pointct： 定义切入点表达式

pointcut-ref： 指定切入点表达式的引用

执行时间点：

切入点方法正常执行之后。它和异常通知只能有一个执行

```
<aop:after-returning method="commit" pointcut-ref="pt1"/>
```

aop:after-throwing

作用：

用于配置异常通知

属性：

method： 指定通知中方法的名称。

pointct： 定义切入点表达式

pointcut-ref： 指定切入点表达式的引用

执行时间点：

切入点方法执行产生异常后执行。它和后置通知只能执行一个

```
<aop:after-throwing method="rollback" pointcut-ref="pt1"/>
```

aop:after

作用：

用于配置最终通知

属性：

method： 指定通知中方法的名称。

pointct： 定义切入点表达式

pointcut-ref： 指定切入点表达式的引用

执行时间点：

无论切入点方法执行时是否有异常，它都会在其后面执行。

```
<aop:after method="release" pointcut-ref="pt1"/>
```

2.2.3 切入点表达式说明

execution:匹配方法的执行(常用)

execution(表达式)

表达式语法：execution([修饰符] 返回值类型 包名.类名.方法名(参数))

写法说明：

全匹配方式：

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

访问修饰符可以省略

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

```
切入点表达式的写法；
    关键字：executing(表达式)
    表达式：访问修饰符 返回值 包名.包名.包名...类名(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：* *..*.saveAccount()和* *..*.*()
```

参数列表：可以直接写数据类型
基本类型可以直接写名称
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，有参数可以是任意参数
全通配写法：* *..*.*(..)
实际开发中，切入点表达式的通常写法：切到业务层实现类的所有方法
* com.itheima.service.impl.*.*(..)

返回值可以使用*号，表示任意返回值

*

com.itheima.service.impl.AccountServiceImpl.saveAccount(com.itheima.domain.Account)

包名可以使用*号，表示任意包，但是有几级包，需要写几个*

* *.*.*.*.AccountServiceImpl.saveAccount(com.itheima.domain.Account)

使用..来表示当前包，及其子包

* com..AccountServiceImpl.saveAccount(com.itheima.domain.Account)

类名可以使用*号，表示任意类

* com..*.saveAccount(com.itheima.domain.Account)

方法名可以使用*号，表示任意方法

* com..*.*(com.itheima.domain.Account)

参数列表可以使用*，表示参数可以是任意数据类型，但是必须有参数

* com..*.*(*)

参数列表可以使用..表示有无参数均可，有参数可以是任意类型

* com..*.*(..)

全通配方式：

* *..*.*(..)

注：

通常情况下，我们都是对业务层的方法进行增强，所以切入点表达式都是切到业务层实现类。

execution(`* com.itheima.service.impl.*.*(..)`**)**

2.2.4 环绕通知

问题：当配置了环绕通知后，切入点的方法没有执行，而通知的方法执行了
分析：通过对比动态代理的环绕通知代码，发现动态代理的环绕通知有明确的切入点方法调用，而此代码没有
解决：Spring框架提供了一个接口: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框架会提供该接口的实现类可以使用
spring中的环绕通知：它是spring框架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

配置方式：

```
<aop:config>
    <aop:pointcut    expression="execution(*    com.itheima.service.impl.*.*(..))"
id="pt1"/>
    <aop:aspect id="txAdvice" ref="txManager">
        <!-- 配置环绕通知 -->
        <aop:around method="transactionAround" pointcut-ref="pt1"/>
    </aop:aspect>
</aop:config>
```

aop:around：

作用：

用于配置环绕通知

属性：

method：指定通知中方法的名称。

pointct：定义切入点表达式

pointcut-ref：指定切入点表达式的引用

说明：

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代码什么时候执行的方式。

注意：

通常情况下，环绕通知都是独立使用的

```
/**
 * 环绕通知
 * @param pjp
 *  spring 框架为我们提供了一个接口：ProceedingJoinPoint，它可以作为环绕通知的方法参数。
 *  在环绕通知执行时，spring 框架会为我们提供该接口的实现类对象，我们直接使用就行。
 * @return
 */
public Object transactionAround(ProceedingJoinPoint pjp) {
    //定义返回值
    Object rtValue = null;
    try {
        //获取方法执行所需的参数
        Object[] args = pjp.getArgs();
        //前置通知：开启事务
        beginTransaction();
        //执行方法
        rtValue = pjp.proceed(args);
        //后置通知：提交事务
        commit();
    }catch(Throwable e) {
        //异常通知：回滚事务
        rollback();
        e.printStackTrace();
    }finally {
        //最终通知：释放资源
        release();
    }
    return rtValue;
}
```

2.3 基于注解的 AOP 配置

2.3.1 环境搭建

2.3.1.1 第一步：准备必要的代码和 jar 包

拷贝上一小节的工程即可。

2.3.1.2 第二步：在配置文件中导入 context 的名称空间

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:context="http://www.springframework.org/schema/context"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   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">

    <!-- 配置数据库操作对象 -->
    <bean id="dbAssit" class="com.itheima.dbassit.DBAssit">
        <property name="dataSource" ref="dataSource"></property>
        <!-- 指定 connection 和线程绑定 -->
        <property name="useCurrentConnection" value="true"></property>
    </bean>

    <!-- 配置数据源 -->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
        <property name="jdbcUrl" value="jdbc:mysql:///spring_day02"></property>
        <property name="user" value="root"></property>
        <property name="password" value="1234"></property>
    </bean>
</beans>
```

2.3.1.3 第三步：把资源使用注解配置

```
/**
 * 账户的业务层实现类
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
@Service("accountService")
public class AccountServiceImpl implements IAccountService {

    @Autowired
    private IAccountDao accountDao;
```

```
}

/**
 * 账户的持久层实现类
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
@Repository("accountDao")
public class AccountDaoImpl  implements IAccountDao {

    @Autowired
    private DBAssit dbAssit ;
}
```

2.3.1.4 第四步：在配置文件中指定 spring 要扫描的包

```
<!-- 告知 spring，在创建容器时要扫描的包 -->
<context:component-scan base-package="com.itheima"></context:component-scan>
```

2.3.2 配置步骤

2.3.2.1 第一步：把通知类也使用注解配置

```
/**
 * 事务控制类
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
@Component("txManager")
public class TransactionManager {
    //定义一个 DBAssit
    @Autowired
    private DBAssit dbAssit ;
 }
```

2.3.2.2 第二步：在通知类上使用@Aspect 注解声明为切面

作用：

把当前类声明为切面类。

```
/**
 * 事务控制类
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
@Component("txManager")
@Aspect//表明当前类是一个切面类
public class TransactionManager {

    //定义一个 DBAssit
    @Autowired
    private DBAssit dbAssit ;
}
```

2.3.2.3 第三步：在增强的方法上使用注解配置通知

@Before

作用：

把当前方法看成是前置通知。

属性：

value：用于指定切入点表达式，还可以指定切入点表达式的引用。

```
//开启事务
@Before("execution(* com.itheima.service.impl.*.*(..))")
public void beginTransaction() {
    try {
        dbAssit.getCurrentConnection().setAutoCommit(false);
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
```

@AfterReturning

作用：

把当前方法看成是后置通知。

属性：

value：用于指定切入点表达式，还可以指定切入点表达式的引用

```
//提交事务
@AfterReturning("execution(* com.itheima.service.impl.*.*(..))")
public void commit() {
```

```
        try {
            dbAssit.getCurrentConnection().commit();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }


    @AfterThrowing
        作用：
            把当前方法看成是异常通知。
        属性：
            value：用于指定切入点表达式，还可以指定切入点表达式的引用
    //回滚事务
    @AfterThrowing("execution(* com.itheima.service.impl.*.*(..))")
    public void rollback() {
        try {
            dbAssit.getCurrentConnection().rollback();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    @After
        作用：
            把当前方法看成是最终通知。
        属性：
            value：用于指定切入点表达式，还可以指定切入点表达式的引用
    //释放资源
    @After("execution(* com.itheima.service.impl.*.*(..))")
    public void release() {
        try {
            dbAssit.releaseConnection();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

2.3.2.4 第四步：在 spring 配置文件中开启 spring 对注解 AOP 的支持

```
    <!-- 开启 spring 对注解 AOP 的支持 -->
    <aop:aspectj-autoproxy/>
```

2.3.3 环绕通知注解配置

@Around

作用：

把当前方法看成是环绕通知。

属性：

value：用于指定切入点表达式，还可以指定切入点表达式的引用。

```
/**
 * 环绕通知
 * @param pjp
 * @return
 */
@Around("execution(* com.itheima.service.impl.*.*(..))")
public Object transactionAround(ProceedingJoinPoint pjp) {
    //定义返回值
    Object rtValue = null;
    try {
        //获取方法执行所需的参数
        Object[] args = pjp.getArgs();
        //前置通知：开启事务
        beginTransaction();
        //执行方法
        rtValue = pjp.proceed(args);
        //后置通知：提交事务
        commit();
    }catch(Throwable e) {
        //异常通知：回滚事务
        rollback();
        e.printStackTrace();
    }finally {
        //最终通知：释放资源
        release();
    }
    return rtValue;
}
```

2.3.4 切入点表达式注解

@Pointcut

作用：

指定切入点表达式

属性：

```
    value：指定表达式的内容

@Pointcut("execution(* com.itheima.service.impl.*.*(..))")
private void pt1() {}
```

引用方式：

```
/**
 * 环绕通知
 * @param pjp
 * @return
 */
@Around("pt1()")//注意：千万别忘了写括号
public Object transactionAround(ProceedingJoinPoint pjp) {
    //定义返回值
    Object rtValue = null;
    try {
        //获取方法执行所需的参数
        Object[] args = pjp.getArgs();
        //前置通知：开启事务
        beginTransaction();
        //执行方法
        rtValue = pjp.proceed(args);
        //后置通知：提交事务
        commit();
    }catch(Throwable e) {
        //异常通知：回滚事务
        rollback();
        e.printStackTrace();
    }finally {
        //最终通知：释放资源
        release();
    }
    return rtValue;
}
```

2.3.5 不使用 XML 的配置方式

```
@Configuration
@ComponentScan(basePackages="com.itheima")
@EnableAspectJAutoProxy
public class SpringConfiguration {
}
```