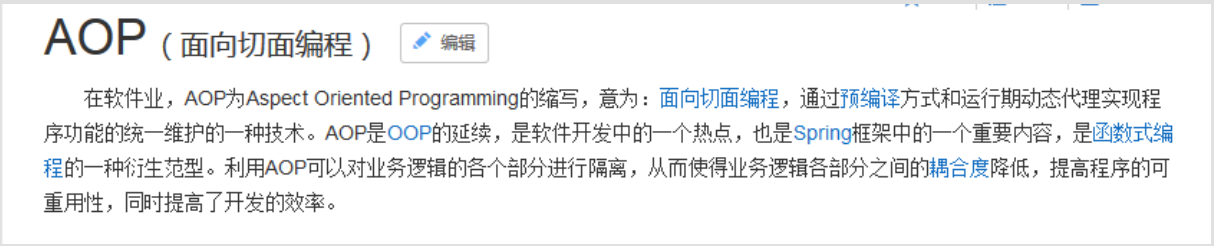
Spring 第三天

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：

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 作业中问题

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);
    Account target = accountDao.findByName(targetName);
    //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);
    int i=1/0; //模拟转账异常
    accountDao.update(target);
}
```

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**）

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：

```
/**
 * 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 AccountServiceImpl2 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 void saveAccount(Account account) {
        try {
            //开启事务
            txManager.beginTransaction();
            accountDao.save(account);
            //提交事务
            txManager.commit();
        }catch(Exception e) {
            //回滚事务
            txManager.rollback();
            throw new RuntimeException(e);
        }finally {
            //释放资源
            txManager.release();
        }
    }

    @Override
    public void updateAccount(Account account) {
        try {
            //开启事务
            txManager.beginTransaction();
            accountDao.update(account);
            //提交事务
            txManager.commit();
        }catch(Exception e) {
            //回滚事务
            txManager.rollback();
            throw new RuntimeException(e);
        }finally {
            //释放资源
            txManager.release();
```

```
        }
    }

    @Override
    public void deleteAccount(Integer accountId) {

        try {
            //开启事务
            txManager.beginTransaction();
            accountDao.delete(accountId);
            //提交事务
            txManager.commit();
        }catch(Exception e) {
            //回滚事务
            txManager.rollback();
            throw new RuntimeException(e);
        }finally {
            //释放资源
            txManager.release();
        }
    }

    @Override
    public List<Account> findAllAccount() {

        try {
            //开启事务
            txManager.beginTransaction();
            List<Account> accounts = accountDao.findAll();
            //提交事务
            txManager.commit();
            return accounts;
        }catch(Exception e) {
            //回滚事务
            txManager.rollback();
            throw new RuntimeException(e);
        }finally {
            //释放资源
            txManager.release();
        }
    }

    @Override
    public Account findById(Integer accountId) {
```

```
        try {
            //开启事务
            txManager.beginTransaction();
            Account account = accountDao.findById(accountId);
            //提交事务
            txManager.commit();
            return account;
        }catch(Exception e) {
            //回滚事务
            txManager.rollback();
            throw new RuntimeException(e);
        }finally {
            //释放资源
            txManager.release();
        }
    }

    @Override
    public void transfer(String sourceName, String targetName, Float money) {
        try {
            //开启事务
            txManager.beginTransaction();

            //1.根据名称查询转出账户
            Account source = accountDao.findAccountByName(sourceName);
            //2.根据名称查询转入账户
            Account target = accountDao.findAccountByName(targetName);
            //3.转出账户减钱
            source.setMoney(source.getMoney()-money);
            //4.转入账户加钱
            target.setMoney(target.getMoney()+money);
            //5.更新转出账户
            accountDao.update(source);
            //模拟转账异常
            int i=1/0;
            //6.更新转入账户
            accountDao.update(target);

            //提交事务
            txManager.commit();
        }catch(Exception e) {
            //回滚事务
            txManager.rollback();
```

```
                throw new RuntimeException(e);
            }finally {
                //释放资源
                txManager.release();
            }
        }

    }
```

TransactionManager 类的代码：

```
/**
 * 事务管理器
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
public class TransactionManager {

    private ConnectionUtil connectionUtil;

    public void setConnectionUtil(ConnectionUtil connectionUtil) {
        this.connectionUtil = connectionUtil;
    }

    //开启事务
    public void beginTransaction() {
        //从当前线程上获取连接，实现开启事务
        try {
            connectionUtil.getThreadConnection().setAutoCommit(false);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    //提交事务
    public void commit() {
        try {
            connectionUtil.getThreadConnection().commit();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    //回滚事务
```

```
    public void rollback() {
        try {
            connectionUtil.getThreadConnection().rollback();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    //释放连接
    public void release() {
        try {
            //关闭连接（还回池中）
            connectionUtil.getThreadConnection().close();
            //解绑线程：把连接和线程解绑
            connectionUtil.remove();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

ConnectionUtils 类代码：

```
/**
 * 一个管理连接的工具类，用于实现连接和线程的绑定
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
public class ConnectionUtil {

    private  ThreadLocal<Connection> tl = new ThreadLocal<Connection>();


    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    /**
     * 获取当前线程上绑定的连接
     * @return
     */
    public Connection getThreadConnection() {
```

```
            try {
                //1.先看看线程上是否绑了
                Connection conn = tl.get();
                if(conn == null) {
                    //2.从数据源中获取一个连接
                    conn = dataSource.getConnection();
                    //3.和线程局部变量绑定
                    tl.set(conn);
                }
                //4.返回线程上的连接
                return tl.get();
            } catch (SQLException e) {
                throw new RuntimeException(e);
            }
        }

        /**
         * 把连接和当前线程解绑
         */
        public void remove() {
            tl.remove();
        }
    }
```

spring 的 xml 配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    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">

    <!-- 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="runner" ref="runner"></property>
        <property name="connectionUtil" ref="connectionUtil"></property>
    </bean>

    <!-- 配置 QueryRunner -->
    <bean id="runner" class="org.apache.commons.dbutils.QueryRunner"></bean>
```

```
    <!-- 配置 ConnectionUtil -->
    <bean id="connectionUtil" class="com.itheima.utils.ConnectionUtil">
        <property name="dataSource" ref="dataSource"></property>
    </bean>

    <!-- 配置自定义事务管理器 -->
    <bean id="txManager" class="com.itheima.utils.TransactionManager">
        <property name="connectionUtil" ref="connectionUtil"></property>
    </bean>

    <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>
```

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

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

此处我们使用的是生产商和经销商的例子：

在早些年，大概 20 年前的样子，我们购买电脑都是去一些大型的商场，这些商场有一块区域专门用于出售电脑，该区域的租金都是由各个生产厂家承租下来，然后雇人销售。但是随着时间的推移，生产厂家发现运营成本过大，希望通过找一些分销团队减轻一些压力，这就是早期的经销商（代理商）。

当然，现在我们看到的一些大型电商平台以及大型的实体卖场（例如：国美，苏宁等）已经很具规模了。

当这些经销机构做大做强之后，对于销售哪些产品，他们也有自己的要求。第一：产品好卖。第二：厂家的售后服务好。

我们接下来，就通过代码模拟一下：

```
/**
 * 一个代理商（经销商）的要求：
 *       有好的产品
 *       有好的售后
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
public interface IProxyProducer {

    /**
     * 销售商品
     * @param money
     */
    public void saleProduct(Float money);

    /**
     * 售后服务
     * @param money
     */
    public void afterService(Float money) ;
}

/**
 * 一个生产厂家
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
```

```
 */
public class Producer implements IProxyProducer{

    /**
     * 销售商品
     * @param money
     */
    public void saleProduct(Float money) {
        System.out.println("销售商品，金额是："+money);
    }

    /**
     * 售后服务
     * @param money
     */
    public void afterService(Float money) {
        System.out.println("提供售后服务，金额是："+money);
    }
}
/**
 * 一个消费者
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
public class Consumer {

    public static void main(String[] args) {
        Producer producer = new Producer();
//      producer.saleProduct(5000f);
//      producer.afterService(1000f);

        /**
         * 动态代理：
         *    特点：字节码随用随创建，随用随加载
         *    分类：基于接口的动态代理，基于子类的动态代理
         *    作用：不修改源码的基础上对方法增强
         *    基于接口的动态代理：
         *        提供者是：JDK 官方
         *        使用要求：被代理类最少实现一个接口。
         *        涉及的类：Proxy
         *        创建代理对象的方法：newProxyInstance
         *        方法的参数：
         *            ClassLoader：类加载器。用于加载代理对象的字节码的。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。用于给代理对象提供方法。和被代理对象具有相同的方法。和被代理对象实现相同的接口，就会具有相同的方法。固定写法
             *          InvocationHanlder：要增强的方法。此处是一个接口，我们需要提供它的实现类。通常写的是匿名内部类。
             *                          增强的代码谁用谁写。
             *
             */
            IProxyProducer proxyProducer = (IProxyProducer) Proxy.newProxyInstance(producer.getClass().getClassLoader(),
                    producer.getClass().getInterfaces(), new InvocationHandler() {
                        /**
                         * 执行被代理对象的任何方法都都会经过该方法，该方法有拦截的作用
                         * 参数的含义
                         *  Object proxy：代理对象的引用。一般不用
                         *  Method method：当前执行的方法
                         *  Object[] args：当前方法所需的参数
                         * 返回值的含义
                         *  和被代理对象的方法有相同的返回值
                         */
                        @Override
                        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                            Object rtValue = null;

                            //1.获取当前执行方法的钱
                            Float money = (Float)args[0];
                            //2.判断当前方法是销售还是售后
                            if("saleProduct".equals(method.getName())) {
                                //销售
                                rtValue = method.invoke(producer, money*0.75f);
                            }
                            if("afterService".equals(method.getName())) {
                                //售后
                                rtValue = method.invoke(producer, money*0.9f);
                            }
                            return rtValue;
                        }
                    });

            proxyProducer.saleProduct(8000f);
            proxyProducer.afterService(1000f);
        }
    }
```

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。

```
/**
 * 一个生产厂家
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
public class Producer{

    /**
     * 销售商品
     * @param money
     */
    public void saleProduct(Float money) {
        System.out.println("销售商品，金额是 cglib: "+money);
    }

    /**
     * 售后服务
     * @param money
     */
    public void afterService(Float money) {
        System.out.println("提供售后服务，金额是 cglib: "+money);
    }
}


/**
 * 一个消费者
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
public class Consumer {

    public static void main(String[] args) {
        Producer producer = new Producer();

        /**
         * 动态代理：
         *    特点：字节码随用随创建，随用随加载
```

```
            *    分类：基于接口的动态代理，基于子类的动态代理
            *    作用：不修改源码的基础上对方法增强
            *    基于子类的动态代理
            *        提供者是：第三方 cglib 包，在使用时需要先导包(maven 工程导入坐标即可)
            *        使用要求：被代理类不能是最终类，不能被 final 修饰
            *        涉及的类：Enhancer
            *        创建代理对象的方法：create
            *        方法的参数：
            *            Class：字节码。被代理对象的字节码。可以创建被代理对象的子类，还可以获取被代理对象的类加载器。
            *            Callback：增强的代码。谁用谁写。通常都是写一个接口的实现类或者匿名内部类。
            *                    我们在使用时一般都是使用 Callback 接口的子接口：MethodInterceptor
            */
           Producer cglibProducer = (Producer)Enhancer.create(producer.getClass(), new MethodInterceptor() {

               /**
                * 执行被代理对象的任何方法都会经过该方法
                * 方法的参数：
                *   前 3 个和基于接口中方法的参数含义一样。
                *   MethodProxy methodProxy：当前执行方法的代理对象。
                * 方法的返回值：
                *   和被代理对象中方法有相同的返回值
                */
               @Override
               public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {
                   Object rtValue = null;

                   //1.获取当前执行方法的钱
                   Float money = (Float)args[0];
                   //2.判断当前方法是销售还是售后
                   if("saleProduct".equals(method.getName())) {
                       //销售
                       rtValue = method.invoke(producer, money*0.5f);
                   }
                   if("afterService".equals(method.getName())) {
                       //售后
                       rtValue = method.invoke(producer, money*0.8f);
                   }
                   return rtValue;
               }
```

```
            });

            cglibProducer.saleProduct(8000f);
            cglibProducer.afterService(1000f);
        }
    }
```

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,
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;
        }
    }
```

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。

2.1.2 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。

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，采用输出日志作为示例。

在业务层方法执行的前后，加入日志的输出。

并且把 spring 的 ioc 也一起应用进来。

2.2.1 第一步：创建 maven 工程并导入坐标

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.0.2.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.8.7</version>
```

```
        </dependency>
    </dependencies>
```

2.2.2 第二步：准备必要的代码

```
/**
 * 账户的业务层接口
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
public interface IAccountService {

    /**
     * 模拟保存
     */
    void saveAccount();

    /**
     * 模拟更新
     * @param i
     */
    void updateAccount(int i);

    /**
     * 模拟删除
     * @return
     */
    int deleteAccount();
}

/**
 * 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 {

    @Override
    public void saveAccount() {
        System.out.println("保存了账户");
    }
```

```
    @Override
    public void updateAccount(int i) {
        System.out.println("更新了账户"+i);
    }

    @Override
    public int deleteAccount() {
        System.out.println("删除了账户");
        return 0;
    }
}

/**
 * 模拟一个用于记录日志的工具类
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
public class Logger {

    /**
     * 用于打印日志
     * 计划让其在切入点方法执行之前执行
     */
    public void printLog() {
        System.out.println("Logger 类中的 printLog 方法开始记录日志了。。。");
    }
}
```

2.2.3 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>
```

2.2.4 第四步：配置 spring 的 ioc

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

2.2.5 第五步：配置 aop

```
<!--
aop 的配置步骤：
    第一步：把通知类的创建也交给 spring 来管理
    第二步：使用 aop:config 标签开始 aop 的配置
    第三步：使用 aop:aspect 标签开始配置切面，写在 aop:config 标签内部
            id 属性：给切面提供一个唯一标识
            ref 属性：用于引用通知 bean 的 id。
    第四步：使用对应的标签在 aop:aspect 标签内部配置通知的类型
            使用 aop:befored 标签配置前置通知，写在 aop:aspect 标签内部
                method 属性：用于指定通知类中哪个方法是前置通知
                pointcut 属性：用于指定切入点表达式。
            切入点表达式写法：
                关键字：execution(表达式)
            表达式内容：
                全匹配标准写法：
                    访问修饰符   返回值   包名.包名.包名...类名.方法名(参数列表)
                例如：
                public void com.itheima.service.impl.AccountServiceImpl.saveAccount()
-->

<!-- 配置通知类 -->
<bean id="logger" class="com.itheima.utils.Logger"></bean>
<!-- 配置 aop -->
<aop:config>
    <!-- 配置切面 -->
    <aop:aspect id="logAdvice" ref="logger">
        <!-- 配置前置通知 -->
        <aop:before          method="printLog"          pointcut="execution(          *
com.itheima.service.impl.*.*(..))"/>
    </aop:aspect>
</aop:config>
```

2.2.6 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)

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.*.*(..))

2.2.7 aop:config

aop:config:

作用：用于声明开始 aop 的配置

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

2.2.8 aop:aspect

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="logAdvice" ref="logger">
        <!--配置通知的类型要写在此处-->
</aop:aspect>
```

2.2.9 aop:pointcut

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(* com.itheima.service.impl.*.*(..))" id="pt1"/>
```

2.2.10通知的四种常用类型

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="beginPrintLog" 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="afterReturningPrintLog" 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="afterThrowingPringLog" 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="afterPringLog" pointcut-ref="pt1"/>
```

2.2.11环绕通知

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
     * 问题：
     *   当配置完环绕通知之后，没有业务层方法执行（切入点方法执行）
     * 分析：
     *  通过动态代理的代码分析，我们现在的环绕通知没有明确的切入点方法调用
     * 解决：
     *   spring 框架为我们提供了一个接口，该接口可以作为环绕通知的方法参数来使用
     *   ProceedingJoinPoint。当环绕通知执行时，spring 框架会为我们注入该接口的实现类。
     *  它有一个方法 proceed()，就相当于 invoke，明确的业务层方法调用
     *
     *  spring 的环绕通知：
     *   它是 spring 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。
     */
    public void aroundPrintLog(ProceedingJoinPoint pjp) {

        try {
            System.out.println("前置 Logger 类中的 aroundPrintLog 方法开始记录日志了");
            pjp.proceed();//明确的方法调用
            System.out.println("后置 Logger 类中的 aroundPrintLog 方法开始记录日志了");
        } catch (Throwable e) {
            System.out.println("异常 Logger 类中的 aroundPrintLog 方法开始记录日志了");
            e.printStackTrace();
        }finally {
            System.out.println("最终 Logger 类中的 aroundPrintLog 方法开始记录日志了");
        }
    }
```

2.3 基于注解的 AOP 配置

2.3.1 第一步：导入 maven 工程的依赖坐标和必要的代码

拷贝上一小节的工程即可。

2.3.2 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:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
```

```
    xmlns:context="http://www.springframework.org/schema/context"
    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 spring 创建容器时要扫描的包 -->
    <context:component-scan base-package="com.itheima"></context:component-scan>
</beans>
```

2.3.3 第三步：把资源使用注解配置

```
@Service("accountService")
public class AccountServiceImpl implements IAccountService {

    @Override
    public void saveAccount() {
        System.out.println("保存了账户");
        int i=1/0;
    }

    @Override
    public void updateAccount(int i) {
        System.out.println("更新了账户"+i);
    }

    @Override
    public int deleteAccount() {
        System.out.println("删除了账户");
        return 0;
    }
}
```

2.3.4 第四步：把通知类也使用注解配置

```
/**
 * 模拟一个用于记录日志的工具类
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
```

```
@Component("logger")
public class Logger {
}
```

2.3.5 第五步：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。

```
/**
 * 模拟一个用于记录日志的工具类
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 * @Version 1.0
 */
@Component("logger")
@Aspect//表示当前类是一个切面类
public class Logger {}
```

2.3.6 第六步：使用注解配置通知类型

@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。

```
/**
 * 前置通知
 */
@Before("execution(* com.itheima.service.impl.*.*(..))")
public void beforePrintLog() {
    System.out.println("前置通知：Logger 类中的 beforePrintLog 方法开始记录日志了。。。");
}
```

@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

```
/**
```

```
         * 后置通知
         */
        @AfterReturning("execution(* com.itheima.service.impl.*.*(..))")
        public void afterReturningPrintLog() {
            System.out.println("后置通知：Logger 类中的 afterReturningPrintLog 方法开始记录日志了。。。");
        }
```

@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

```
        /**
         * 异常通知
         */
        @AfterThrowing("execution(* com.itheima.service.impl.*.*(..))")
        public void afterThrowingPrintLog() {
            System.out.println("异常通知：Logger 类中的 afterThrowingPrintLog 方法开始记录日志了。。。");
        }
```

@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

```
        /**
         * 最终通知
         */
        @After("execution(* com.itheima.service.impl.*.*(..))")
        public void afterPrintLog() {
            System.out.println("最终通知：Logger 类中的 afterPrintLog 方法开始记录日志了。。。");
    }
```

2.3.7 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/>
```

2.3.8 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
 * 问题：
 *   当配置完环绕通知之后，没有业务层方法执行（切入点方法执行）
 * 分析：
 *  通过动态代理的代码分析，我们现在的环绕通知没有明确的切入点方法调用
 * 解决：
 *   spring 框架为我们提供了一个接口，该接口可以作为环绕通知的方法参数来使用
 *   ProceedingJoinPoint。当环绕通知执行时，spring 框架会为我们注入该接口的实现类。
 *  它有一个方法 proceed()，就相当于 invoke，明确的业务层方法调用
 *
 *  spring 的环绕通知：
 *   它是 spring 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。
 */
@Around("execution(* com.itheima.service.impl.*.*(..))")
public void aroundPrintLog(ProceedingJoinPoint pjp) {

    try {
        System.out.println("前置 Logger 类中的 aroundPrintLog 方法开始记录日志了");
        pjp.proceed();//明确的方法调用
        System.out.println("后置 Logger 类中的 aroundPrintLog 方法开始记录日志了");
    } catch (Throwable e) {
        System.out.println("异常 Logger 类中的 aroundPrintLog 方法开始记录日志了");
        e.printStackTrace();
    }finally {
        System.out.println("最终 Logger 类中的 aroundPrintLog 方法开始记录日志了");
    }
}
```

2.3.9 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 void aroundPrintLog(ProceedingJoinPoint pjp) {
}
```

2.3.10不使用 XML 的配置方式

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

第3章 作业

3.1 需 求

使用今天所学 spring 的 aop 通过配置的方式实现对事务的控制，从而去除业务层中的重复代码。

3.2 要 求

实现两个版本的配置

第一个版本：基于 XML 的的配置，使用四种常用通知类型

第二个版本：基于注解的配置，使用环绕通知