

任务二：AOP

课程任务主要内容：

- * 转账案例
- * Proxy优化转账案例
- * 初识AOP
- * 基于XML的AOP开发
- * 基于注解的AOP开发
- * AOP优化转账案例

一 转账案例

需求

使用spring框架整合DBUtils技术，实现用户转账功能

1.1 基础功能

步骤分析

1. 创建java项目，导入坐标
2. 编写Account实体类
3. 编写AccountDao接口和实现类
4. 编写AccountService接口和实现类
5. 编写spring核心配置文件
6. 编写测试代码

1) 创建java项目，导入坐标

```
<dependencies>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
  </dependency>
  <dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.1.15</version>
  </dependency>
  <dependency>
    <groupId>commons-dbutils</groupId>
    <artifactId>commons-dbutils</artifactId>
    <version>1.6</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
```

```

        <artifactId>spring-context</artifactId>
        <version>5.1.5.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.1.5.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
</dependencies>

```

2) 编写Account实体类

```

public class Account {

    private Integer id;
    private String name;
    private Double money;

    // setter getter....

}

```

3) 编写AccountDao接口和实现类

```

public interface AccountDao {

    // 转出操作
    public void out(String outUser, Double money);

    // 转入操作
    public void in(String inUser, Double money);

}

```

```

@Repository
public class AccountDaoImpl implements AccountDao {

    @Autowired
    private QueryRunner queryRunner;

    @Override
    public void out(String outUser, Double money) {
        try {
            queryRunner.update("update account set money=money-? where name=?",
money, outUser);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

}

```

```

@Override
public void in(String inUser, Double money) {
    try {
        queryRunner.update("update account set money=money+? where name=?",
money, inUser);
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}

```

4) 编写AccountService接口和实现类

```

public interface AccountService {

    public void transfer(String outUser, String inUser, Double money);
}

```

```

@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao accountDao;

    @Override
    public void transfer(String outUser, String inUser, Double money) {

        accountDao.out(outUser, money);
        accountDao.in(inUser, money);
    }
}

```

5) 编写spring核心配置文件

```

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

    <!--开启组件扫描-->
    <context:component-scan base-package="com.lagou"/>

    <!--加载jdbc配置文件-->
    <context:property-placeholder location="classpath:jdbc.properties"/>

    <!--把数据库连接池交给IOC容器-->

```

```

<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
    <property name="driverClassName" value="${jdbc.driver}"></property>
    <property name="url" value="${jdbc.url}"></property>
    <property name="username" value="${jdbc.username}"></property>
    <property name="password" value="${jdbc.password}"></property>
</bean>

<!--把QueryRunner交给IOC容器-->
<bean id="queryRunner" class="org.apache.commons.dbutils.QueryRunner">
    <constructor-arg name="ds" ref="dataSource"></constructor-arg>
</bean>
</beans>

```

6) 编写测试代码

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class AccountServiceTest {

    @Autowired
    private AccountService accountService;

    @Test
    public void testTransfer() throws Exception {
        accountService.transfer("tom", "jerry", 100d);
    }
}

```

7) 问题分析

上面的代码事务在dao层，转出转入操作都是一个独立的事务，但实际开发，应该把业务逻辑控制在一个事务中，所以应该将事务挪到service层。

3.2 传统事务

步骤分析

1. 编写线程绑定工具类
2. 编写事务管理器
3. 修改service层代码
4. 修改dao层代码

1) 编写线程绑定工具类

```

/**
 * 连接工具类，从数据源中获取一个连接，并将实现和线程的绑定
 */
@Component
public class ConnectionUtils {

```

```

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

@Autowired
private DataSource dataSource;

/**
 * 获取当前线程上的连接
 *
 * @return Connection
 */
public Connection getThreadConnection() {
    // 1.先从ThreadLocal上获取
    Connection connection = threadLocal.get();
    // 2.判断当前线程是否有连接
    if (connection == null) {
        try {
            // 3.从数据源中获取一个连接，并存入到ThreadLocal中
            connection = dataSource.getConnection();
            threadLocal.set(connection);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    return connection;
}

/**
 * 解除当前线程的连接绑定
 */
public void removeThreadConnection() {
    threadLocal.remove();
}
}

```

2) 编写事务管理器

```

/**
 * 事务管理器工具类，包含：开启事务、提交事务、回滚事务、释放资源
 */
@Component
public class TransactionManager {

    @Autowired
    private ConnectionUtils connectionUtils;

    public void beginTransaction() {
        try {
            connectionUtils.getThreadConnection().setAutoCommit(false);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public void commit() {
        try {

```

```

        connectionUtils.getThreadConnection().commit();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

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

public void release() {
    try {
        connectionUtils.getThreadConnection().setAutoCommit(true); // 改回自
动提交事务
        connectionUtils.getThreadConnection().close(); // 归还到连接池
        connectionUtils.removeThreadConnection(); // 解除线程绑定
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}
}

```

3) 修改service层代码

```

@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao accountDao;

    @Autowired
    private TransactionManager transactionManager;

    @Override
    public void transfer(String outUser, String inUser, Double money) {
        try {
            // 1.开启事务
            transactionManager.beginTransaction();
            // 2.业务操作
            accountDao.out(outUser, money);
            int i = 1 / 0;
            accountDao.in(inUser, money);
            // 3.提交事务
            transactionManager.commit();
        } catch (Exception e) {
            e.printStackTrace();
            // 4.回滚事务
            transactionManager.rollback();
        } finally {

```

```

        // 5.释放资源
        transactionManager.release();
    }
}
}

```

4) 修改dao层代码

```

@Repository
public class AccountDaoImpl implements AccountDao {

    @Autowired
    private QueryRunner queryRunner;

    @Autowired
    private ConnectionUtils connectionUtils;

    @Override
    public void out(String outUser, Double money) {
        try {
            queryRunner.update(connectionUtils.getThreadConnection(), "update
account set money=money-? where name=?", money, outUser);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void in(String inUser, Double money) {
        try {
            queryRunner.update(connectionUtils.getThreadConnection(), "update
account set money=money+? where name=?", money, inUser);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

5) 问题分析

上面代码，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，违背了面向对象的开发思想。

二 Proxy优化转账案例

我们可以将业务代码和事务代码进行拆分，通过动态代理的方式，对业务方法进行事务的增强。这样就不会对业务层产生影响，解决了耦合性的问题啦！

常用的动态代理技术

JDK 代理：基于接口的动态代理技术：利用拦截器（必须实现InvocationHandler）加上反射机制生成一个代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理，从而实现方法增强

CGLIB代理：基于父类的动态代理技术：动态生成一个要代理的子类，子类重写要代理的类的所有不是final的方法。在子类中采用方法拦截技术拦截所有的父类方法的调用，顺势织入横切逻辑，对方法进行增强



2.1 JDK动态代理方式

Jdk工厂类

```
@Component
public class JdkProxyFactory {

    @Autowired
    private AccountService accountService;

    @Autowired
    private TransactionManager transactionManager;

    public AccountService createAccountServiceJdkProxy() {
        AccountService accountServiceProxy = null;
        accountServiceProxy = (AccountService)
        Proxy.newProxyInstance(accountService.getClass().getClassLoader(),
            accountService.getClass().getInterfaces(), new
        InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
                Object result = null;
                try {
                    // 1.开启事务
                    transactionManager.beginTransaction();
                    // 2.业务操作
                    result = method.invoke(accountService, args);
                    // 3.提交事务
                    transactionManager.commit();
                } catch (Exception e) {
                    e.printStackTrace();
                    // 4.回滚事务
                    transactionManager.rollback();
                }
            }
        });
        return accountServiceProxy;
    }
}
```



```

        } finally {
            // 5.释放资源
            transactionManager.release();
        }
        return result;
    }
});
return accountServiceProxy;
}
}

```

测试代码

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class AccountTest {

    @Autowired
    private JdkProxyFactory jdkProxyFactory;

    @Test
    public void testTransfer() throws Exception {
        AccountService accountServiceJdkProxy =
            jdkProxyFactory.createAccountServiceJdkProxy();
        accountServiceJdkProxy.transfer("tom", "jerry", 100d);
    }
}

```

2.2 CGLIB动态代理方式

Cglib工厂类

```

@Component
public class CglibProxyFactory {
    @Autowired
    private AccountService accountService;

    @Autowired
    private TransactionManager transactionManager;

    public AccountService createAccountServiceCglibProxy() {
        AccountService accountServiceProxy = null;

        /*
         * 参数一：目标对象的字节码对象
         * 参数二：动作类，实现增强功能
         */
        accountServiceProxy = (AccountService)
            Enhancer.create(accountService.getClass(), new MethodInterceptor() {
                @Override
            }
        }
    }
}

```

```

        public Object intercept(Object o, Method method, Object[] objects,
MethodProxy methodProxy) throws Throwable {
            Object result = null;
            try {
                // 1.开启事务
                transactionManager.beginTransaction();
                // 2.业务操作
                result = method.invoke(accountService, objects);
                // 3.提交事务
                transactionManager.commit();
            } catch (Exception e) {
                e.printStackTrace();
                // 4.回滚事务
                transactionManager.rollback();
            } finally {
                // 5.释放资源
                transactionManager.release();
            }
            return result;
        }
    });
    return accountServiceProxy;
}
}

```

测试代码

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class AccountServiceTest {

    @Autowired
    private CglibProxyFactory cglibProxyFactory;

    @Test
    public void testTransfer() throws Exception {
        AccountService accountServiceCglibProxy =

        cglibProxyFactory.createAccountServiceCglibProxy();
        accountServiceCglibProxy.transfer("tom", "jerry", 100d);
    }
}

```

三 初识AOP

3.1 什么是AOP

AOP 为 Aspect Oriented Programming 的缩写，意思为**面向切面编程**

AOP 是 OOP（面向对象编程）的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

这样做的好处是：

1. 在程序运行期间，在不修改源码的情况下对方法进行功能增强
2. 逻辑清晰，开发核心业务的时候，不必关注增强业务的代码
3. 减少重复代码，提高开发效率，便于后期维护

3.2 AOP底层实现

实际上，AOP 的底层是通过 Spring 提供的**动态代理技术**实现的。在运行期间，Spring通过动态代理技术动态的生成代理对象，代理对象方法执行时进行增强功能的介入，在去调用目标对象的方法，从而完成功能的增强。

3.3 AOP相关术语

Spring 的 AOP 实现底层就是对上面的动态代理的代码进行了封装，封装后我们只需要对需要关注的部分进行代码编写，并通过配置的方式完成指定目标的方法增强。

在正式讲解 AOP 的操作之前，我们必须理解 AOP 的相关术语，常用的术语如下：

- * **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
- * **Joinpoint**（连接点）：所谓连接点是指那些可以被拦截到的点。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
分类：前置通知、后置通知、异常通知、最终通知、环绕通知
- * **Aspect**（切面）：是切入点和通知（引介）的结合
- * **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

3.4 AOP开发明确事项

3.4.1 开发阶段（我们做的）

1. 编写核心业务代码（目标类的目标方法） 切入点
2. 把公用代码抽取出来，制作成通知（增强功能方法） 通知
3. 在配置文件中，声明切入点与通知间的关系，即切面

3.4.2 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。

3.4.3 底层代理实现

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。

- 当bean实现接口时，会用JDK代理模式
- 当bean没有实现接口，用cglib实现（可以强制使用cglib（在spring配置中加入<aop:aspectj-autoproxy proxy-target-class="true"/>）

3.5 知识小结

- * aop: 面向切面编程
- * aop底层实现：基于JDK的动态代理 和 基于cglib的动态代理
- * aop的重点概念：
 - Pointcut（切入点）：真正被增强的方法
 - Advice（通知/ 增强）：封装增强业务逻辑的方法
 - Aspect（切面）：切点+通知
 - weaving（织入）：将切点与通知结合，产生代理对象的过程

四 基于XML的AOP开发

4.1 快速入门

步骤分析

1. 创建java项目，导入AOP相关坐标
2. 创建目标接口和目标实现类（定义切入点）
3. 创建通知类及方法（定义通知）
4. 将目标类和通知类对象创建权交给spring
5. 在核心配置文件中配置织入关系，及切面
6. 编写测试代码

4.1.1 创建java项目，导入AOP相关坐标

```
<dependencies>
  <!--导入spring的context坐标，context依赖aop-->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.5.RELEASE</version>
  </dependency>
  <!-- aspectj的织入（切点表达式需要用到该jar包） -->
  <dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.13</version>
  </dependency>
  <!--spring整合junit-->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
```

```

        <version>5.1.5.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
</dependencies>

```

4.1.2 创建目标接口和目标实现类

```

public interface AccountService {

    public void transfer();

}

```

```

public class AccountServiceImpl implements AccountService {

    @Override
    public void transfer() {
        System.out.println("转账业务...");
    }

}

```

4.1.3 创建通知类

```

public class MyAdvice {

    public void before() {
        System.out.println("前置通知...");
    }

}

```

4.1.4 将目标类和通知类对象创建权交给spring

```

<!-- 目标类交给IOC容器-->
<bean id="accountService" class="com.lagou.service.impl.AccountServiceImpl">
</bean>
<!-- 通知类交给IOC容器-->
<bean id="myAdvice" class="com.lagou.advice.MyAdvice"></bean>

```

4.1.5 在核心配置文件中配置织入关系，及切面

导入aop命名空间

```

<?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"
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">

<!-- 目标类交给IOC容器-->
<bean id="accountService" class="com.lagou.service.impl.AccountServiceImpl">
</bean>
<!-- 通知类交给IOC容器-->
<bean id="myAdvice" class="com.lagou.advice.MyAdvice"></bean>

<aop:config>
    <!-- 引入通知类-->
    <aop:aspect ref="myAdvice">
        <!-- 配置目标类的transfer方法执行时，使用通知类的before方法进行前置增强-->
        <aop:before method="before"
            pointcut="execution(public void
com.lagou.service.impl.AccountServiceImpl.transfer())"></aop:before>
    </aop:aspect>
</aop:config>
</beans>

```

4.1.6 编写测试代码

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
class AccountServiceTest {

    @Autowired
    private AccountService accountService;

    @Test
    public void testTransfer() throws Exception {
        accountService.transfer();
    }
}

```

4.2 XML配置AOP详解

4.2.1 切点表达式

表达式语法：

```
execution([修饰符] 返回值类型 包名.类名.方法名(参数))
```

- 访问修饰符可以省略
- 返回值类型、包名、类名、方法名可以使用星号 * 代替，代表任意
- 包名与类名之间一个点 . 代表当前包下的类，两个点 .. 表示当前包及其子包下的类
- 参数列表可以使用两个点 .. 表示任意个数，任意类型的参数列表

例如：

```
execution(public void com.lagou.service.impl.AccountServiceImpl.transfer())

execution(void com.lagou.service.impl.AccountServiceImpl.*(..))

execution(* com.lagou.service.impl.*.*(..))

execution(* com.lagou.service..*.*(..))
```

切点表达式抽取

当多个增强的切点表达式相同时，可以将切点表达式进行抽取，在增强中使用 pointcut-ref 属性代替 pointcut 属性来引用抽取后的切点表达式。

```
<aop:config>
  <!--抽取的切点表达式-->
  <aop:pointcut id="myPointcut" expression="execution(* com.lagou.service..*.*(..))"> </aop:pointcut>
  <aop:aspect ref="myAdvice">
    <aop:before method="before" pointcut-ref="myPointcut"></aop:before>
  </aop:aspect>
</aop:config>
```

4.2.2 通知类型

通知的配置语法：

```
<aop:通知类型 method="通知类中方法名" pointcut="切点表达式"></aop:通知类型>
```

名称	标签	说明
前置通知	<aop:before>	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
后置通知	<aop:afterReturning>	用于配置后置通知。指定增强的方法在切入点方法之后执行
异常通知	<aop:afterThrowing>	用于配置异常通知。指定增强的方法出现异常后执行
最终通知	<aop:after>	用于配置最终通知。无论切入点方法执行时是否有异常，都会执行
环绕通知	<aop:around>	用于配置环绕通知。开发者可以手动控制增强代码在什么时候执行

注意：通常情况下，环绕通知都是独立使用的

4.3 知识小结

- * aop织入的配置

```
<aop:config>
    <aop:aspect ref="通知类">
        <aop:before method="通知方法名称" pointcut="切点表达式"></aop:before>
    </aop:aspect>
</aop:config>
```
- * 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
- * 切点表达式
`execution([修饰符] 返回值类型 包名.类名.方法名(参数))`

五 基于注解的AOP开发

5.1 快速入门

步骤分析

1. 创建java项目，导入AOP相关坐标
2. 创建目标接口和目标实现类（定义切入点）
3. 创建通知类（定义通知）
4. 将目标类和通知类对象创建权交给spring
5. 在通知类中使用注解配置织入关系，升级为切面类
6. 在配置文件中开启组件扫描和 AOP 的自动代理
7. 编写测试代码

5.1.1 创建java项目，导入AOP相关坐标

```
<dependencies>
    <!--导入spring的context坐标，context依赖aop-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.1.5.RELEASE</version>
    </dependency>
    <!-- aspectj的织入 -->
    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.8.13</version>
    </dependency>
    <!--spring整合junit-->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.1.5.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
```



```
<artifactId>junit</artifactId>
<version>4.12</version>
</dependency>
</dependencies>
```

5.1.2 创建目标接口和目标实现类

```
public interface AccountService {

    public void transfer();
}
```

```
public class AccountServiceImpl implements AccountService {

    @Override
    public void transfer() {
        System.out.println("转账业务...");
    }
}
```

5.1.3 创建通知类

```
public class MyAdvice {

    public void before() {
        System.out.println("前置通知...");
    }
}
```

5.1.4 将目标类和通知类对象创建权交给spring

```
@Service
public class AccountServiceImpl implements AccountService {}

@Component
public class MyAdvice {}
```

5.1.5 在通知类中使用注解配置织入关系，升级为切面类

```

@Component
@Aspect
public class MyAdvice {

    @Before("execution(* com.lagou..*.*(..))")
    public void before() {
        System.out.println("前置通知...");
    }
}

```

5.1.6 在配置文件中开启组件扫描和 AOP 的自动代理

```

<!--组件扫描-->
<context:component-scan base-package="com.lagou"/>

<!--aop的自动代理-->
<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

```

5.1.7 编写测试代码

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
class AccountServiceTest {

    @Autowired
    private AccountService accountService;

    @Test
    public void testTransfer() throws Exception {
        accountService.transfer();
    }
}

```

5.2 注解配置AOP详解

5.2.1 切点表达式

切点表达式的抽取

```

@Component
@Aspect
public class MyAdvice {

    @Pointcut("execution(* com.lagou...*(..))")
    public void myPoint(){}

    @Before("MyAdvice.myPoint()")
    public void before() {
        system.out.println("前置通知...");
    }
}

```

5.2.2 通知类型

通知的配置语法：@通知注解(“切点表达式”)

名称	标签	说明
前置通知	@Before	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
后置通知	@AfterReturning	用于配置后置通知。指定增强的方法在切入点方法之后执行
异常通知	@AfterThrowing	用于配置异常通知。指定增强的方法出现异常后执行
最终通知	@After	用于配置最终通知。无论切入点方法执行时是否有异常，都会执行
环绕通知	@Around	用于配置环绕通知。开发者可以手动控制增强代码在什么时候执行

注意：

当前四个通知组合在一起时，执行顺序如下：

@Before -> @After -> @AfterReturning (如果有异常：@AfterThrowing)

5.2.3 纯注解配置

```

@Configuration
@ComponentScan("com.lagou")
@EnableAspectJAutoProxy //替代 <aop:aspectj-autoproxy />
public class SpringConfig {

}

```

5.3 知识小结

- * 使用@Aspect注解，标注切面类
- * 使用@Before等注解，标注通知方法
- * 使用@Pointcut注解，抽取切点表达式
- * 配置aop自动代理 <aop:aspectj-autoproxy/> 或 @EnableAspectJAutoProxy

六 AOP优化转账案例

依然使用前面的转账案例，将两个代理工厂对象直接删除！改为spring的aop思想来实现

6.1 xml配置实现

1) 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"
       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">

    <!--开启组件扫描-->
    <context:component-scan base-package="com.lagou"/>

    <!--加载jdbc配置文件-->
    <context:property-placeholder location="classpath:jdbc.properties"/>

    <!--把数据库连接池交给IOC容器-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="${jdbc.driver}"></property>
        <property name="url" value="${jdbc.url}"></property>
        <property name="username" value="${jdbc.username}"></property>
        <property name="password" value="${jdbc.password}"></property>
    </bean>

    <!--把QueryRunner交给IOC容器-->
    <bean id="queryRunner" class="org.apache.commons.dbutils.QueryRunner">
        <constructor-arg name="ds" ref="dataSource"></constructor-arg>
    </bean>

    <!--AOP配置-->
    <aop:config>
        <!--切点表达式-->
```

```

        <aop:pointcut id="myPointcut" expression="execution(*
com.lagou.service..*.*(..))"/>
        <!-- 切面配置 -->
        <aop:aspect ref="transactionManager">
            <aop:before method="beginTransaction" pointcut-ref="myPointcut"/>
            <aop:after-returning method="commit" pointcut-ref="myPointcut"/>
            <aop:after-throwing method="rollback" pointcut-ref="myPointcut"/>
            <aop:after method="release" pointcut-ref="myPointcut"/>
        </aop:aspect>
    </aop:config>

</beans>

```

2) 事务管理器 (通知)

```

// 事务管理器工具类, 包括: 开启事务、提交事务、回滚事务、释放资源
@Component
public class TransactionManager {

    @Autowired
    ConnectionUtils connectionUtils;

    public void begin(){
        try {
            connectionUtils.getThreadConnection().setAutoCommit(false);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public void commit(){
        try {
            connectionUtils.getThreadConnection().commit();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

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

    public void release(){
        try {
            connectionUtils.getThreadConnection().setAutoCommit(true);
            connectionUtils.getThreadConnection().close();
            connectionUtils.removeThreadConnection();
        } catch (SQLException e) {

```

```

        e.printStackTrace();
    }
}
}

```

6.2 注解配置实现

1) 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"
       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">

    <!--开启组件扫描-->
    <context:component-scan base-package="com.lagou"/>

    <!--开启AOP注解支持-->
    <aop:aspectj-autoproxy/>

    <!--加载jdbc配置文件-->
    <context:property-placeholder location="classpath:jdbc.properties"/>

    <!--把数据库连接池交给IOC容器-->
    <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">
        <property name="driverClassName" value="${jdbc.driver}"></property>
        <property name="url" value="${jdbc.url}"></property>
        <property name="username" value="${jdbc.username}"></property>
        <property name="password" value="${jdbc.password}"></property>
    </bean>

    <!--把QueryRunner交给IOC容器-->
    <bean id="queryRunner" class="org.apache.commons.dbutils.QueryRunner">
        <constructor-arg name="ds" ref="dataSource"></constructor-arg>
    </bean>

</beans>

```

2) 事务管理器（通知）

```

@Component
@Aspect

```

```
public class TransactionManager {

    @Autowired
    ConnectionUtils connectionUtils;

    @Around("execution(* com.lagou.serivce..*.*(..))")
    public Object around(ProceedingJoinPoint pjp) {
        Object object = null;

        try {
            // 开启事务
            connectionUtils.getThreadConnection().setAutoCommit(false);

            // 业务逻辑
            pjp.proceed();

            // 提交事务
            connectionUtils.getThreadConnection().commit();

        } catch (Throwable throwable) {
            throwable.printStackTrace();
            // 回滚事务
            try {
                connectionUtils.getThreadConnection().rollback();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }

        } finally {
            try {
                connectionUtils.getThreadConnection().setAutoCommit(true);
                connectionUtils.getThreadConnection().close();
                connectionUtils.removeThreadConnection();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        return object;
    }
}
```