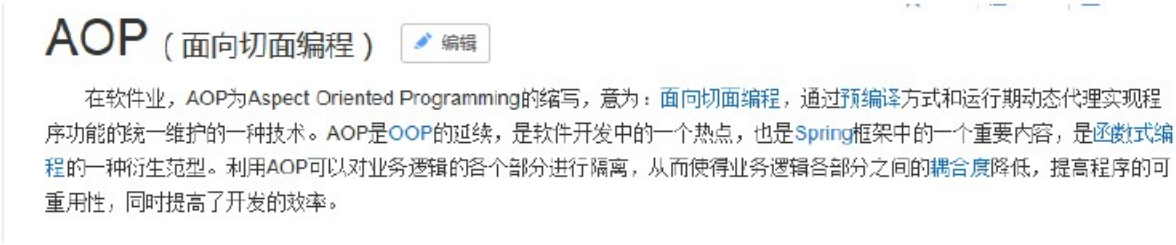
Spring 第三天

第1章 AOP 的相关概念

1.1 AOP 概述

1.1.1 什么是 AOP

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


AOP（面向切面编程） 编辑

在软件业，AOP为Aspect Oriented Programming的缩写，意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是OOP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，是函数式编程的一种衍生范型。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。

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

可能通过上面的介绍，我们还是没有一个清晰的认识。没关系，我们看看下面的具体应用。

1.2 AOP 的具体应用

1.2.1 案例中问题

这是我们之前在 struts2 课程中做的一个完整的增删改查例子。下面是客户的业务层接口和实现类。

通过下面的代码，我们能看出什么问题吗?

客户的业务层接口

```
/**
 * 客户的业务层接口
 */
public interface ICustomerService {

    /**
     * 保存客户
     * @param customer
     */
    void saveCustomer(Customer customer);

    /**
     * 查询所有客户
     * @return
     */
    List<Customer> findAllCustomer();

    /**
     * 删除客户
     * @param customer
     */
    void removeCustomer(Customer customer);

    /**
     * 根据 id 查询客户
     * @param custId
     * @return
     */
    Customer findCustomerById(Long custId);

    /**
     * 修改客户
     * @param customer
     */
    void updateCustomer(Customer customer);
}
```

客户的业务层实现类

```
/**
 * 客户的业务层实现类
 * 事务必须在此控制
 * 业务层都是调用持久层的方法
```

```
 */
public class CustomerServiceImpl implements ICustomerService {

    private ICustomerDao customerDao = new CustomerDaoImpl();

    @Override
    public void saveCustomer(Customer customer) {
        Session s = null;
        Transaction tx = null;
        try{
            s = HibernateUtil.getCurrentSession();
            tx = s.beginTransaction();
            customerDao.saveCustomer(customer);
            tx.commit();
        }catch(Exception e){
            tx.rollback();
            throw new RuntimeException(e);
        }
    }

    @Override
    public List<Customer> findAllCustomer() {
        Session s = null;
        Transaction tx = null;
        try{
            s = HibernateUtil.getCurrentSession();
            tx = s.beginTransaction();
            List<Customer> customers = customerDao.findAllCustomer();
            tx.commit();
            return customers;
        }catch(Exception e){
            tx.rollback();
            throw new RuntimeException(e);
        }
    }

    @Override
    public void removeCustomer(Customer customer) {
        Session s = null;
        Transaction tx = null;
        try{
            s = HibernateUtil.getCurrentSession();
            tx = s.beginTransaction();
            customerDao.removeCustomer(customer);
```

```
            tx.commit();
        }catch(Exception e){
            tx.rollback();
            throw new RuntimeException(e);
        }
    }

    @Override
    public Customer findCustomerById(Long custId) {
        Session s = null;
        Transaction tx = null;
        try{
            s = HibernateUtil.getCurrentSession();
            tx = s.beginTransaction();
            Customer c = customerDao.findCustomerById(custId);
            tx.commit();
            return c;
        }catch(Exception e){
            tx.rollback();
            throw new RuntimeException(e);
        }
    }

    @Override
    public void updateCustomer(Customer customer) {
        Session s = null;
        Transaction tx = null;
        try{
            s = HibernateUtil.getCurrentSession();
            tx = s.beginTransaction();
            customerDao.updateCustomer(customer);
            tx.commit();
        }catch(Exception e){
            tx.rollback();
            throw new RuntimeException(e);
        }
    }
}
```

上面代码的问题就是：我们的事务控制是重复性的代码。这还只是一个业务类，如果有多个业务了，每个业务类中都会有这些重复性的代码。

思考：

我们有什么办法解决这个问题吗？

要想解决这个问题，我们得回顾一下之前我们讲过的知识：动态代理。

1.2.2 动态代理回顾

1.2.2.1 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。

1.2.2.2 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）。

1.2.2.3 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;
                        //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);
        }
```

```
}
```

1.2.2.4 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);
                        }
                    }
                    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);
        }
    }
```

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？

1.2.3 解决案例中的问题

思路只有一个：使用动态代理技术创建客户业务层的代理对象，在执行CustomerServiceImpl时，对里面的方法进行增强，加入事务的支持。

```
    /**
     * 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）
    */
    public class BeanFactory {

        /**
         * 获取客户业务层对象的代理对象
         * @return
         */
```

```
        public static ICustomerService getCustomerService() {
            //定义客户业务层对象
            final ICustomerService customerService = new CustomerServiceImpl();
            //生成它的代理对象
            ICustomerService proxyCustomerService = (ICustomerService)

    Proxy.newProxyInstance(customerService.getClass().getClassLoader()
                ,customerService.getClass().getInterfaces(),
                new InvocationHandler() {
                //执行客户业务层任何方法，都会在此处被拦截，我们对那些方法增强，加入事务。

                @Override
                public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {
                    String name = method.getName();
                    Object rtValue = null;
                    try{
                        //开启事务
                        HibernateUtil.beginTransaction();
                        //执行操作
                        rtValue = method.invoke(customerService, args);

                        //提交事务
                        HibernateUtil.commit();
                    }catch(Exception e){
                        //回滚事务
                        HibernateUtil.rollback();
                        e.printStackTrace();
                    }finally{
                        //释放资源.hibernate 在我们事务操作（提交/回滚）之后，已经帮我
们关了。
                        //如果他没关，我们在此处关
                    }
                    return rtValue;
                }
            });
            return proxyCustomerService;
        }
    }
```

1.3 Spring 中的 AOP

1.3.1 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。

1.3.2 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。

1.3.3 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。

第2章 基于 XML 的 AOP 配置

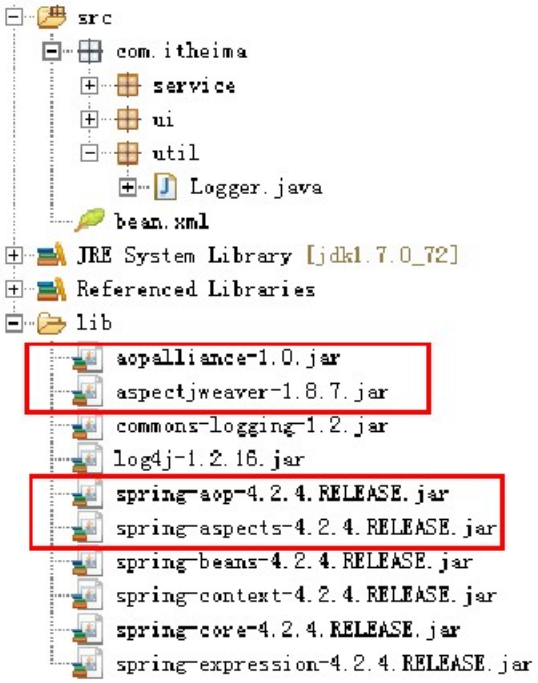
2.1 环境搭建

2.1.1 第一步：准备客户的业务层和接口（需要增强的类）

```
/**
 * 客户的业务层接口
*/
public interface ICustomerService {

    /**
     * 保存客户
     */
    void saveCustomer();

    /**
     * 修改客户
     * @param i
     */
    void updateCustomer(int i);
}

/**
 * 客户的业务层实现类
 */
public class CustomerServiceImpl implements ICustomerService {

    @Override
    public void saveCustomer() {
        System.out.println("调用持久层，执行保存客户");
    }

    @Override
    public void updateCustomer(int i) {
        System.out.println("调用持久层，执行修改客户");
    }
```

```
    }
```

2.1.2 第二步：拷贝必备的 jar 包到工程的 lib 目录



2.1.3 第三步：创建 spring 的配置文件并导入约束

```
    <?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>
```

2.1.4 第四步：把客户的业务层配置到 spring 容器中

```
    <!-- 把资源交给 spring 来管理 -->
    <bean                                                id="customerService"
class="com.itheima.service.impl.CustomerServiceImpl"/>
```

2.1.5 第五步：制作通知（增强的类）

```
/**
 * 一个记录日志的工具类
*/
public class Logger {
    /**
     * 期望：此方法在业务核心方法执行之前，就记录日志
     */
    public void beforePrintLog(){
        System.out.println("Logger 类中的 printLog 方法开始记录日志了。。。。");
    }
}
```

2.2 配置步骤

2.2.1 第一步：把通知类用 bean 标签配置起来

```
<!-- 把有公共代码的类也让 spring 来管理（把通知类也交给 spring 来管理） -->
<bean id="logger" class="com.itheima.util.Logger"></bean>
```

2.2.2 第二步：使用 aop:config 声明 aop 配置

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

2.2.3 第三步：使用 aop:aspect 配置切面

```
<!-- 配置切面 ：此标签要出现在 aop:config 内部
    id：给切面提供一个唯一标识
    ref：引用的是通知类的 bean 的 id
-->
<aop:aspect id="logAdvice" ref="logger">
        <!--配置通知的类型要写在此处-->
</aop:aspect>
```

2.2.4 第四步：使用 aop:before 配置前置通知

```
    <!-- 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
            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
    -->
    <aop:before method="beforePrintLog" pointcut-ref="pt1"/>
```

2.2.5 第五步：使用 aop:pointcut 配置切入点表达式

```
    <aop:pointcut                expression="execution(public                    void
com.itheima.service.impl.CustomerServiceImpl.saveCustomer())"
    id="pt1"/>
```

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.CustomerServiceImpl.saveCustomer()
        访问修饰符可以省略
            void com.itheima.service.impl.CustomerServiceImpl.saveCustomer()
        返回值可以使用*号，表示任意返回值
            * com.itheima.service.impl.CustomerServiceImpl.saveCustomer()
        包名可以使用*号，表示任意包，但是有几级包，需要写几个*
            * *.*.*.*.CustomerServiceImpl.saveCustomer()
        使用..来表示当前包，及其子包
            * com..CustomerServiceImpl.saveCustomer()
        类名可以使用*号，表示任意类
            * com..*.saveCustomer()
        方法名可以使用*号，表示任意方法
            * com..*.*()
        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：
            * *..*.*(..)
```

2.4 常用标签

2.4.1 <aop:config>

作用：

用于声明开始 aop 的配置

2.4.2 <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。

2.4.3 <aop:pointcut>

作用：

用于配置切入点表达式

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。

2.4.4 <aop:before>

作用：

用于配置前置通知

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

2.4.5 <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

2.4.6 <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

2.4.7 <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

2.4.8 <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

2.5 通知的类型

2.5.1 类型说明

```
<!-- 配置通知的类型
    aop:before:
        用于配置前置通知。前置通知的执行时间点：切入点方法执行之前执行
    aop:after-returning:
        用于配置后置通知。后置通知的执行时间点：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-throwing
```

```
            用于配置异常通知。异常通知的执行时间点：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
            用于配置最终通知。最终通知的执行时间点：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:around
            用于配置环绕通知。他和前面四个不一样，他不是用于指定通知方法何时执行的。
    -->
    <aop:before method="beforePrintLog" pointcut-ref="pt1"/>
    <aop:after-returning                    method="afterReturningPrintLog" pointcut-ref="pt1"/>
    <aop:after-throwing method="afterThrowingPrintLog" pointcut-ref="pt1"/>
    <aop:after method="afterPrintLog" pointcut-ref="pt1"/>
    <aop:around method="aroundPringLog" pointcut-ref="pt1"/>
```

2.5.2 环绕通知的特殊说明

```
    /**
         * 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部分什么时候执行的方式。
         * 问题：
         *  当我们配置了环绕通知之后，增强的代码执行了，业务核心方法没有执行。
         * 分析：
         *  通过动态代理我们知道在 invoke 方法中，有明确调用业务核心方法：method.invoke()。
         *  我们配置的环绕通知中，没有明确调用业务核心方法。
         * 解决：
         *  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。
         *  该接口中有一个方法 proceed()，此方法就相当于 method.invoke()
         */
        public void aroundPringLog(ProceedingJoinPoint pjp){
            try {
                System.out.println("前置通知：Logger 类的 aroundPringLog 方法记录日志");
                pjp.proceed();
                System.out.println("后置通知：Logger 类的 aroundPringLog 方法记录日志");
            } catch (Throwable e) {
                System.out.println("异常通知：Logger 类的 aroundPringLog 方法记录日志");
                e.printStackTrace();
```

```
            }finally{
                System.out.println("最终通知：Logger 类的 aroundPringLog 方法记录日志");
            }
        }
```

第3章 基于注解的 AOP 配置

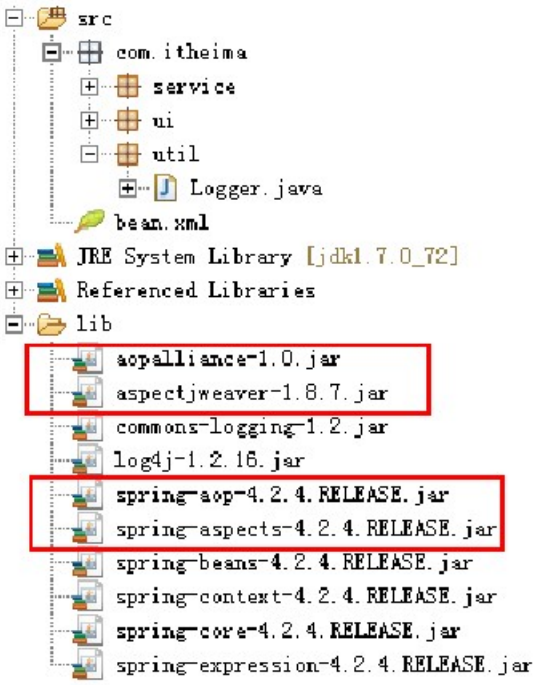
3.1 环境搭建

3.1.1 第一步：准备客户的业务层和接口并用注解配置（需要增强的类）

```
/**
 * 客户的业务层接口
*/
public interface ICustomerService {

    /**
     * 保存客户
     */
    void saveCustomer();

    /**
     * 修改客户
     * @param i
     */
    void updateCustomer(int i);
}

/**
 * 客户的业务层实现类
 */
public class CustomerServiceImpl implements ICustomerService {

    @Override
    public void saveCustomer() {
        System.out.println("调用持久层，执行保存客户");
    }
```

```
    @Override
    public void updateCustomer(int i) {
        System.out.println("调用持久层，执行修改客户");
    }
}
```

3.1.2 第二步：拷贝必备的 jar 包到工程的 lib 目录



3.1.3 第三步：创建 spring 的配置文件并导入约束

```
<?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"
       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">

</beans>
```

3.1.4 第四步：把资源使用注解让 spring 来管理

```
/**
 * 客户的业务层实现类
 */
@Service("customerService")
public class CustomerServiceImpl implements ICustomerService {

    @Override
    public void saveCustomer() {
        System.out.println("调用持久层，执行保存客户");
    }
    @Override
    public void updateCustomer(int i) {
        System.out.println("调用持久层，执行修改客户");
    }
}
```

3.1.5 第五步：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>
```

3.2 配置步骤

3.2.1 第一步：把通知类也使用注解配置

```
/**
 * 一个记录日志的工具类
 */
@Component("logger")
public class Logger {
    /**
     * 期望：此方法在业务核心方法执行之前，就记录日志
     * 前置通知
     */
    public void beforePrintLog(){
        System.out.println("前置通知:Logger 类中的 printLog 方法开始记录日志了");
    }
}
```

3.2.2 第二步：在通知类上使用@Aspect 注解声明为切面

```
/**
 * 一个记录日志的工具类
 */
@Component("logger")
@Aspect//表明当前类是一个切面类
public class Logger {
    /**
     * 期望：此方法在业务核心方法执行之前，就记录日志
     * 前置通知
     */
    public void beforePrintLog(){
        System.out.println("前置通知:Logger 类中的 printLog 方法开始记录日志了");
    }
}
```

3.2.3 第三步：在增强的方法上使用@Before 注解配置前置通知

```
    /**
     * 期望：此方法在业务核心方法执行之前，就记录日志
     * 前置通知
     */
    @Before("execution(* com.itheima.service.impl.*.*(..))")//表示前置通知
    public void beforePrintLog(){
        System.out.println("前置通知:Logger 类中的 printLog 方法开始记录日志了");
    }
```

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/>
```

3.3 常用注解

3.3.1 @Aspect：

```
作用：
    把当前类声明为切面类。
```

3.3.2 @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。

3.3.3 @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。

3.3.4 @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。

3.3.5 @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。

3.3.6 @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。

3.3.7 @Pointcut

作用：
　　指定切入点表达式
属性：

```
value：指定表达式的内容
```

3.4 不使用 XML 的配置方式

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