第3章 Spring AOP

本章学习内容

- Spring AOP 概述；
- 动态代理模式；
- AspectJ 表达式；
- 使用 XML 配置实现 AOP；
- 使用注解实现 AOP。

3.1 Spring AOP 概述

3.1.1 AOP 的概念

AOP 是 Spring 框架中最核心的一个功能，首先通过一个生活中的案例来学习一下什么是 AOP。比如银行系统会有一个取款流程，传统程序的流程如图 3-1 所示。

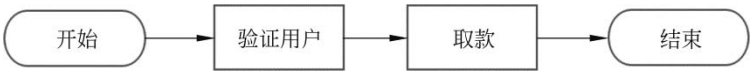

图 3-1 取款流程

假设系统还有一个查询并显示余额的流程，如图 3-2 所示。

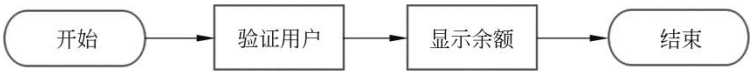

图 3-2 显示余额流程

把这两个流程放到一起，会发现两者有一个相同的验证流程，如图 3-3 所示。

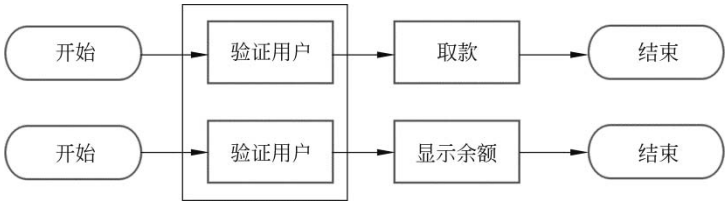

图 3-3 验证流程

在系统开发中，验证用户的功能是相同的，但又在不同的地方出现，可以把这部分代码提取出来写成一个共用的类，这个类就称为切面（Aspect）类。切面类中的方法（验证用户）称为通知（Advice），在程序执行时动态地添加到主业务类的方法（取款和显示余额）前后，这个主业务类的方法称为连接点（Joinpoint）。验证用户的切面类可以在不同的地方重用，这种编程方式叫作 AOP（Aspect Oriented Programming），即面向切面编程。

有了 AOP，在编写代码时就不需要把验证用户的步骤写进去，可以完全不考虑验证用户的功能，只编写取款和显示余额的业务代码，而在另一个地方写好验证用户的代码。这个验证用户的代码就是切面代码，以后在执行取款和显示余额时，将验证用户的功能在执行取款和显示余额前调用。

代码在 Spring 容器中执行时，通过配置告诉 Spring 这段代码要添加的地方，Spring 就会在执行正常业务流程时把验证代码和取款代码织入在一起。

AOP 的真正目的是让程序员在编写代码时只需考虑主要业务流程，而不用考虑和业务无关却又必须要写的代码。

3.1.2 AOP 中类与切面的关系

AOP 的本质是在一系列纵向的控制流程中，把那些相同的子流程（如验证用户）提取成一个横向的面，将分散在主流程中相同的代码提取出来，然后在程序编译或运行时，将这些提取出来的切面代码应用到需要执行的地方。

例如取款、查询、转账前都要进行用户验证，则验证用户就可以做成切面类。在执行取款、查询、转账的操作时，由 Spring 容器将验证用户的代码织入在它们的前面，从而达到验证用户的目的。验证用户的代码只需要编写一次，可以让程序员将编程的精力放在取款、查询、转账等主要业务上。切面与主业务类的关系如图 3-4 所示。

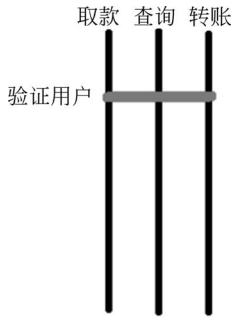

图 3-4 切面与主业务类的关系

3.1.3 AOP 的应用场景

面向切面编程的主要场景有如下几种，切面可以分别在类 1 和类 2 方法中加入事务、日志、权限控制等功能，如图 3-5 所示。上面的验证用户就是权限控制的一种。

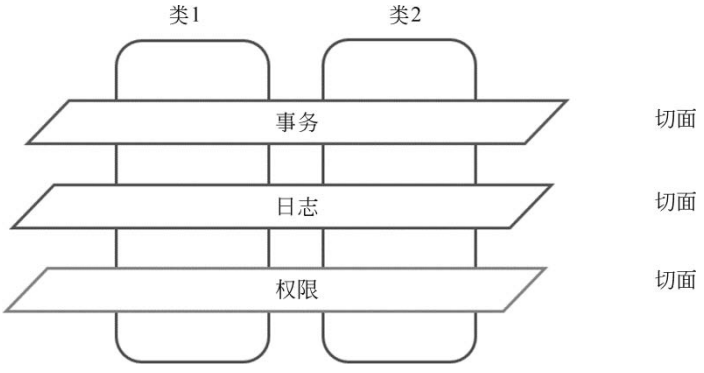

图 3-5 AOP 的应用场景

比如日志记录功能，服务器端重要的操作步骤是需要用日志记录下来，便于以后服务器的管理和维护，所以系统中就会出现类似如下代码。

```
logger.info("管理员登录");          //日志记录
userService.login();               //业务操作
logger.info("管理员删除用户");      //日志记录
userService.deleteUser();          //业务操作
logger.info("管理员退出");          //日志记录
userService.logout();              //业务操作
```

上面的业务代码和日志记录代码会分布在整个系统中，而且是零散的。

几乎所有的重要操作方法前面都会加上日志记录代码，这样的代码写起来烦琐，不但占用了开发时间和精力，而且不容易维护。因此可以将日志记录的功能做成切面类，让程序在执行时再动态地将日志与主业务功能织入在一起。

AOP 的核心实现是动态代理模式，下面介绍一下 Java 中常用的 JDK 动态代理。

3.2 动态代理模式

代理模式的作用是为其他对象提供一种代理以便控制该对象的访问。代理模式可以详细控制访问某个对象的方法，在调用这个方法前做一些前置处理，调用这个方法后也可以做后置处理。代理模式的实现分为静态代理和动态代理，在 Spring 中多使用动态代理。动态代理的实现可以分为 JDK 动态代理和 CGLIB 动态代理，下文重点介绍 JDK 动态代理。

3.2.1 代理模式对象

JDK 动态代理等代理模式所涉及的对象如图 3-6 所示，如租客、中介、房东等都是代理对象。

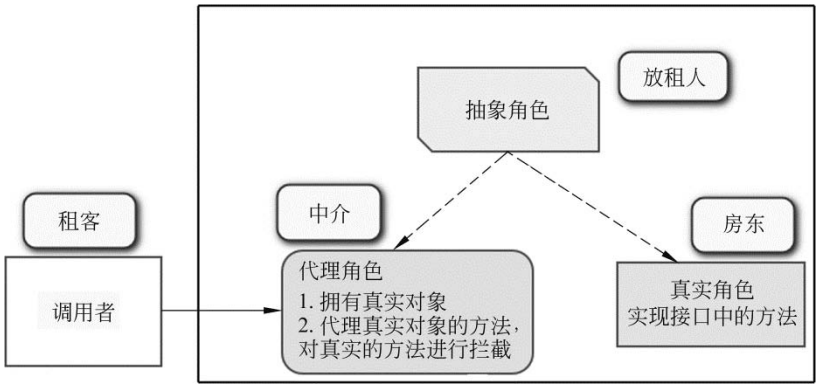

图 3-6 代理模式示意图

在代理模式中有以下 4 种角色。

（1）真实角色：需要实现抽象角色的接口，定义了真实角色所要实现的业务逻辑，即真正的业务逻辑。

（2）代理角色：相当于真实角色的一个代理角色，可以改写真实角色的方法或对真实角色的方法进行拦截，并可以附加自己的操作。

（3）抽象角色：指代理角色和真实角色对外提供的公共方法，定义了真实角色的行为，一般为一个抽象类或接口。

（4）调用者：使用真实角色的调用者，不属于代理模式中的一部分。

3.2.2 JDK 动态代理


视频讲解

JDK 动态代理必须借助于一个接口才能产生代理对象。因此，对于使用业务接口的类，Spring 默认使用 JDK 动态代理实现 AOP。

1. 动态代理的特点

（1）程序在执行过程中动态生成代理对象，不用手动编写代理对象。

（2）不需要重写目标对象中所有同名的方法，只关注需要代理的方法即可。

2. 动态代理类相应的 API

1）Proxy 类

```
    static Object newProxyInstance(ClassLoader loader, Class[] interfaces,
InvocationHandler h)
```

在 JDK 的 API 中存在一个 Proxy 类，其中有一个生成动态代理对象的方法 newProxyInstance()。

该类的参数说明如下。

（1）loader：真实对象的类加载器。

（2）interfaces：真实对象所有实现的接口数组。

（3）h：具体的代理操作，InvocationHandler 是一个接口，需要传入一个实现了此接口的实现类。

（4）返回值：生成的代理对象。

2）InvocationHandler 接口

```
    Object invoke(Object proxy, Method method, Object[] args)
```

在这个方法中实现对真实方法的增强或拦截，其参数说明如下。

（1）proxy：即 newProxyInstance()方法返回的代理对象，该对象一般不在 invoke 方法中使用，容易出现递归调用。

（2）method：真实对象的方法对象，会进行多次调用，每次调用 method 对象都不同。

（3）args：代理对象调用方法时传递的参数。

（4）返回值：真实对象方法的返回值。

下面通过一个案例演示如何使用 JDK 动态代理实现 Spring AOP。

假设有一个出租的接口，其代码如下。

```
    package com.ssmbook2020;
    /**
     * 出租的接口
     */
    public interface Lease {
        /**
         * 定义出租的行为
         * @param money 租金
```

```
     */
    void rentOut(int money);
}
```

真实角色是房东，其代码如下。

```
/**
 * 房东：真实角色
 */
public class Landlord implements Lease {
    @Override
    public void rentOut(int money){
        System.out.println("房东出租房子，收取租金：" + money);
    }
}
```

该房屋出租案例的开发流程如下。

（1）直接创建真实对象，并调用真实对象的方法。

（2）通过 Proxy 类创建代理对象，并调用代理对象的方法。

（3）分别输出真实对象和代理对象的实现类。

```
package com.ssmbook2020;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
/**
 * 租客：使用代理对象的调用者
 */
public class Tenant {
    public static void main(String[] args){
        //(1)直接找房东租房(创建真实对象)
        Lease landlord = new Landlord();
        //调用真实对象的方法
        landlord.rentOut(2000);
        System.out.println("真实对象：" + landlord.getClass());
        //输出一条线分隔开
        System.out.println("==========");
        //(2)找中介租房 (创建代理对象)
        Lease agent = (Lease)
        Proxy.newProxyInstance(landlord.getClass(). getClassLoader();
 //真实对象的类加载器
                landlord.getClass().getInterfaces();
//获取真实对象所有实现的接口
                new InvocationHandler() { //代理的实现
                    /**
                     * proxy：即 newProxyInstance()方法返回的代理对象
                     * method: 真实对象的方法对象，会被调用多次，每次调用 method 对象
                     * 都不同
```

```
                     * args：代理对象调用方法时传递的参数
                     */
                    @Override
          public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable{
                        //如果是出租的方法
                        if (method.getName().equals("rentOut")){
                            //对真实方法进行代理，但不修改原来类的方法
    System.out.println("中介出租房子，收取中介费：200");
                        }
                        //调用真实对象的方法
                        return method.invoke(landlord, args);
                    }
                });
          //调用代理对象的方法
          agent.rentOut(2000);
          //输出代理对象
          System.out.println("代理对象：" + agent.getClass());
       }

    }
```

在以上代码中，代理角色是程序在执行过程中动态生成的，在没有修改真实对象的前提下对真实对象进行了代理，并添加了新的功能。

该案例的代码执行效果如下。

```
房东出租房子，收取租金：2000
真实对象：class com.ssmbook2020.Landlord
==========
中介出租房子，收取中介费：200
房东出租房子，收取租金：2000
代理对象：class com.sun.proxy.$Proxy0
```

可以看到，代理对象实现类的名字是 com.sun.proxy.$Proxy0，在后面的 Spring AOP 中还会再用到。

3.3 AOP 的实现

视频讲解

3.3.1 AOP 的常用增强类型

AOP 的实现原理其实就是使用代理模式，由 AOP 框架动态生成一个代理对象，该对象可作为代替目标对象使用。Spring 中的 AOP 代理由 Spring 的 IoC 容器负责生成、管理，因此 AOP 代理可以直接使用容器中的其他 Bean 实例作为目标，这种关系可由 IoC 容器的依赖注入提供。

常用的 AOP 代理增强主要包括前置增强、后置增强、环绕增强、异常增强等。

（1）前置增强（Before Advice）：在某连接点之前执行的增强，但这个增强不能阻止连接点之前的执行流程（除非它抛出一个异常）。

（2）后置增强（After Returning Advice）：在某连接点正常完成后执行的增强。例如，一个方法没有抛出任何异常，正常返回。

（3）异常增强（After Throwing Advice）：在方法抛出异常退出时执行的增强。

（4）最终增强（After (Finally) Advice）：当某连接点退出时执行的增强（无论是正常返回还是异常退出）。

（5）环绕增强（Around Advice）：包围一个连接点的增强，如方法调用。这是最强大的一种增强类型。环绕增强可以在方法调用前后完成自定义的行为，它会选择是否继续执行连接点、直接返回它自己的返回值或抛出异常来结束执行。

在内部调用这 5 种增强类型的组织方式如下。

```
try {
调用前置增强
环绕前置处理
调用目标对象方法
环绕后置处理
调用后置增强
} catch(Exception e) {
调用异常增强
} finally {
调用最终增强
}
```

视频讲解

3.3.2 AspectJ 表达式

AspectJ 表达式又称为切入点表达式，它其实是一组规则，指定哪些包下的哪些类和方法织入通知代码。

1. 切点函数

常用的切点函数有以下 3 个，如表 3-1 所示。

表 3-1 切点函数

切点函数	作　用
execution	细粒度函数，可以精确到类中的某个方法
within	粗粒度函数，只能精确到类
bean	粗粒度函数，只能精确到类，它是通过 id 从容器中获取对象

2. 表达式语法

图 3-7 是 Spring 官方文档对表达式语法的介绍，其中？号表示出现 0 次或 1 次。

切点函数中共有 6 个参数可以指定，只有“返回类型”、“方法名”和“参数列表”是必须指定的，参数中可以使用通配符，不同位置的通配符的含义有所区别。

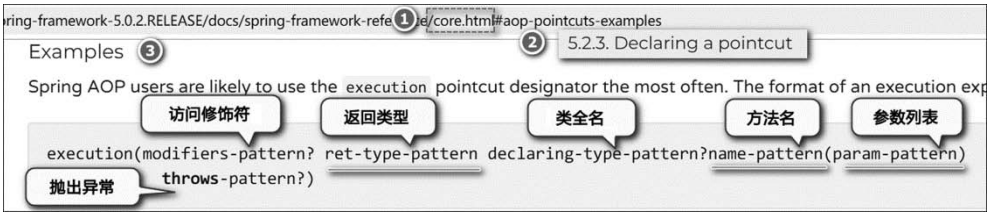

图 3-7　切点函数的语法

- 方法中参数个数通配符写法：

() 表示没有参数。

(*) 表示 1 个任意类型的参数。

(..) 表示 0 个或多个参数。

- 类全名的包通配符写法：

.. 表示当前包和子包。

3. 举例说明

（1）最精确的写法。

```
    execution(public void com.ssmbook2020.service.impl.AccountServiceImpl.
save())
```

（2）匹配 Service 的包和子包下面所有的类和方法，方法参数是 String 类型。

```
    execution(* com.ssmbook2020.service..*.*(String))
```

（3）覆盖最全的写法，匹配所有的类和方法。

```
    execution(* *(..))
```

（4）匹配方法名是 save 或 update 的方法。匹配时也可以使用&&符号，虽然语法是正确的，但没有意义。

```
    execution(* save(..)) || execution(* update(..))
```

（5）除了方法名是 save 的所有方法。

```
    !execution(* save(..))
```

（6）匹配包和子包中的所有类，只精确到类。

```
    within(com.itheima..*)
```

（7）从容器中获取一个 id 为 accountService 的类中的所有方法，只精确到类。

```
    bean(accountService)
```

（8）从容器中获取所有 Service 的方法，只精确到类。

```
    bean(*Service)
```

3.3.3 使用 XML 配置方式实现 AOP

在 Spring 中，AOP 编程可以使用配置或注解两种实现方式，下面对此分别进行介绍。首先通过一个案例来介绍 XML 配置方式，该案例实现的功能是：当向数据库中保存账户时，使用日志记录这次保存操作。

1. 开发流程

该案例的开发步骤如下。

（1）开发业务类，主业务方法为添加账户。

（2）开发切面类，用于在每个主业务方法执行前添加记录日志的功能。

（3）使用 AOP 将业务类与切面类织入在一起，实现需求的功能。

2. 案例结构

整个案例的工程结构如图 3-8 所示。

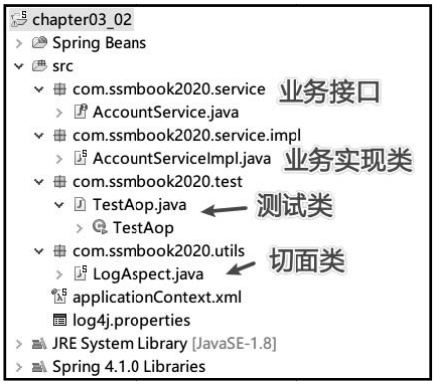

图 3-8 XML 配置方式的案例结构

3. 业务接口和实现类

（1）建立账户业务接口并保存账户。

```
package com.ssmbook2020.service;
/**
 * 账户业务接口
 */
public interface AccountService {
    /**
     * 保存账户
     */
    void save();
}
```

（2）实现业务接口类，输出“保存账户”。

```
package com.ssmbook2020.service.impl;
import com.ssmbook2020.service.AccountService;
```

```
/**
 * 实现类
 */
public class AccountServiceImpl implements AccountService {
    @Override
    public void save() {
        System.out.println("保存账户");
    }
}
```

4. 记录日志的工具类

```
package com.ssmbook2020.utils;
import java.sql.Timestamp;
/**
 * 记录日志功能的类：切面类 = 切入点(规则)+通知(方法)
 */
public class LogAspect {
    /**
     * 记录日志
     */
    public void printLog() {
        System.out.println(new Timestamp(System.currentTimeMillis()) + " 记录日志");
    }
}
```

5. AOP 的配置

1）AOP 配置结构图

图 3-9 为 Spring AOP 的配置结构图。在一个项目中只需配置一次。读者在编写 XML 配置时可以参照图 3-9 进行编写，避免出错。

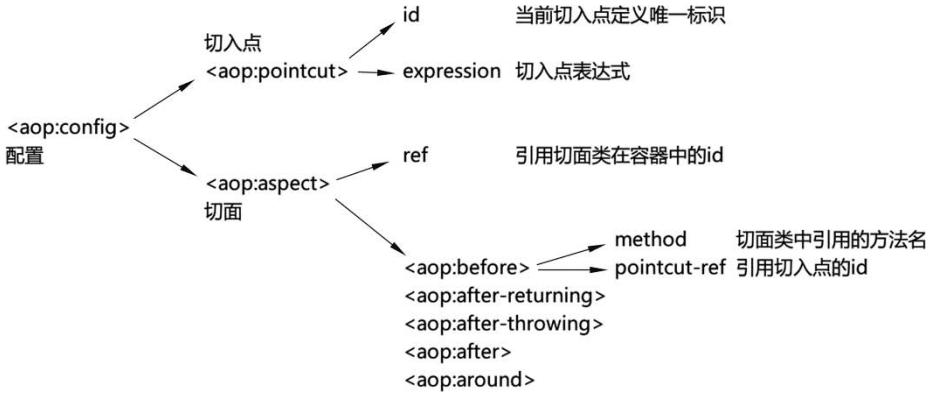

图 3-9　AOP 的配置结构

2）配置文件

接下来在 xml 配置文件中编写 AOP 的配置，主要步骤如下。

（1）配置日志记录类，包括切面类 LogAspect。

（2）配置正常的业务类 AccountServiceImpl。

（3）进行 AOP 配置，配置流程参考图 3-9。需要注意的是，XML 的 Schema 需要导入 AOP 的命名空间。

```
<?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:p="http://www.springframework.org/schema/p"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.1.xsd
    http://www.springframework.org/schema/aop
    https://www.springframework.org/schema/aop/spring-aop-4.1.xsd">
     <!--正常业务对象-->
    <bean class="com.ssmbook2020.service.impl.AccountServiceImpl" id=
"accountService"/>
    <!-- 切面类：日志记录对象 -->
    <bean class="com.ssmbook2020.utils.LogAspect" id="logAspect"/>

    <!-- 编写 Aop 的配置，要导入 AOP 的命名空间 -->
    <aop:config>
        <!--
            配置切入点，通过切入点表达式来配置
             id：给表达式定义唯一标识
             expression: 使用切入点函数定义表达式，语法为“访问修饰符 返回类型 包名.
类名.方法名(参数类型) 抛出异常类型” -->
        <aop:pointcut id="pt" expression="execution(public void
com.ssmbook2020.service.impl.AccountServiceImpl.save())"/>

        <!-- 切面配置， ref 引用切面对象 id -->
        <aop:aspect ref="logAspect">
            <!-- 使用什么类型的通知：前置通知，后置通知等
            method：表示切面中的方法名字 pointcut-ref：引用上面的切入点表达式 -->
            <aop:before method="printLog" pointcut-ref="pt"/>
        </aop:aspect>
    </aop:config>
</beans>
```

3）测试类

配置完成 AOP 后，开始编写测试类，其步骤如下。

（1）调用业务方法。

（2）输出业务类的 getClass()，查看输出的代理类对象。

```
package com.ssmbook2020.test;
```

```
    import org.springframework.context.support.ClassPathXmlApplicationContext;
    import com.ssmbook2020.service.AccountService;
    public class TestAop {
        public static void main(String[] args){
            ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
            //从容器中获取对象
            AccountService accountService = context.getBean(AccountService.class);
            //得到的是代理对象
            System.out.println(accountService.getClass());
            //调用业务方法
            accountService.save();
            //关闭容器
            context.close();
        }
    }
```

测试类的运行结果如下。

```
    class com.sun.proxy.$Proxy5
    2021-01-20 12:22:11.373 记录日志
    保存账户
```

AOP 的配置本质上是使用代理模式实现功能增强，不需要自己编写代理模式，而通过配置就可以实现。由此可以看到，此时的 AccountService 的实现类其实是个代理对象。

4）执行流程分析

因为配置了 AOP，分开编写的切面类 LogAspect 和正常业务类 AccountServiceImpl 被 Spring AOP 框架在执行时织入在一起，生成了一个代理对象，如图 3-10 所示。

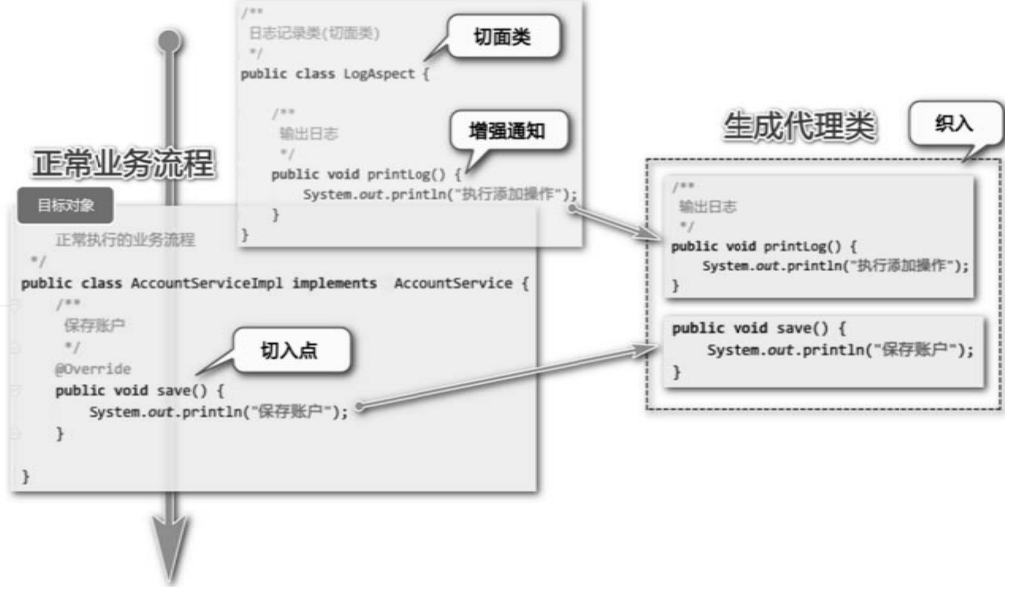

图 3-10　织入的流程

Spring AOP 其实使用了两种动态代理实现方式，如果一个类并没有实现任何的接口，则无法使用上面所说的 JDK 动态代理，这时需要使用 CGLIB 代理。CGLIB 代理的本质是对原有类的继承，子类重写相应的方法，其生成过程与 JDK 类似，这里不再赘述。


视频讲解

3.3.4 使用注解方式实现 AOP

通过 3.3.3 节的案例可以知道，使用 XML 的方式配置 AOP 是比较烦琐的，如何简化 XML 的配置呢？可以使用注解的方式。

对 Spring AOP 中常用的 Aspect 注解进行如下介绍。

（1）@Aspect：放在类的上面，表示这是一个切面类。

（2）@Before：前置增强。它有以下两个参数。

- value：该成员用于定义切点。
- execution：切点函数，告诉 Spring 在哪些地方进行前置增强的织入。

（3）@AfterReturning：后置增强。它有以下 4 个参数。

- value: 该成员用于定义切点。
- pointcut：表示切点的信息。如果指定 pointcut 值，将覆盖 value 的值，可以理解为它们的作用是相同的。
- returning：将目标对象方法的返回值绑定给增强的方法，返回值的名字要与实际返回的变量名相同。
- argNames：同 returning。

（4）@Around：环绕增强。它有 value 和 argNames 两个参数，其含义同上。

（5）@AfterThrowing：异常增强，有以下 4 个参数。

- value、pointcut、argNames 同上。
- throwing：将抛出的异常绑定到增强方法中。

（6）@After：最终增强。不管是抛出异常或是正常退出，该增强都会得到执行，它有 Value 和 arg Names 两个参数，其含义同上。

1. 案例需求

案例的业务需求描述如下。

在登录的方法前面输出日志。用户开始登录，在登录方法的后面输出提示日志：用户登录成功或失败。

在业务实现过程中，使用注解定义前置增强和后置增强，从而实现日志功能。

2. 开发步骤

该案例的主要步骤如下。

（1）创建 Java 项目，添加 Spring 框架，注解方式的项目结构如图 3-11 所示。

（2）编写各个类的代码如下，注意代码注释。

- User.java 用户实体类。

```
package com.ssmbook2020.entity;
/**
```

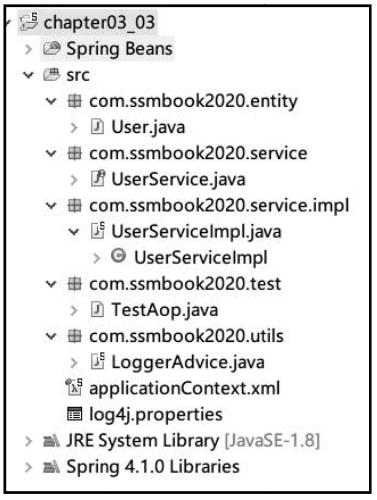

图 3-11　注解方式的项目结构

```
 * 用户实体类对象
 */
public class User {
    private int id;           //主键
    private String name;      //用户名
    private String password;  //密码

    public User(int id, String name, String password) { //带全参的构造方法
        super();
        this.id = id;
        this.name = name;
        this.password = password;
    }

    public User() {           //默认无参的构造方法
        super();
    }
    //省略了 get 和 set 方法
}
```

- UserService.java 业务接口类。

```
package com.ssmbook2020.service;
import com.ssmbook2020.entity.User;
/**
 * 用户的业务接口
 */
public interface UserService {
    /**
     * 登录的方法
     * @param name 用户名
```

```
         * @param password 密码
         * @return 登录成功返回User对象，登录失败返回null
         */
        public User login(String name,String password);
    }
```

- UserServiceImpl.java 业务实现类。

```
    package com.ssmbook2020.service.impl;
    import com.ssmbook2020.entity.User;
    import com.ssmbook2020.service.UserService;
    /**
     * 用户业务类的实现
    */
    public class UserServiceImpl implements UserService {
        @Override
        public User login(String name, String password){
            System.out.println("业务方法login运行，正在登录...");
            //登录成功
            if ("newboy".equals(name) && "520".equals(password)){
                return new User(100, "newboy", "520");
            }
            //登录失败
            return null;
        }
    }
```

- LoggerAdvice.java 切面类。

```
    package com.ssmbook2020.utils;
    import java.sql.Timestamp;
    import org.apache.log4j.Logger;
    import org.aspectj.lang.JoinPoint;
    import org.aspectj.lang.annotation.AfterReturning;
    import org.aspectj.lang.annotation.Aspect;
    import org.aspectj.lang.annotation.Before;

    /** 需要织入的日志切面类 */
    @Aspect
    public class LoggerAdvice {
        //log4j日志类
        Logger logger = Logger.getLogger(LoggerAdvice.class);

        //后置增强
        @AfterReturning(pointcut = "execution(* com.ssmbook2020..*.*(..))",
returning = "ret")
        public void afterReturning(JoinPoint join, Object ret){
            String method = join.getSignature().getName();
```

```
            Object[] args = join.getArgs();
            if ("login".equals(method)){
                if (ret != null){
                    logger.info(new Timestamp(System.currentTimeMillis()) + " "
+ args[0] + "登录成功");
                } else {
                    logger.info(new Timestamp(System.currentTimeMillis()) + " "
+ args[0] + "登录失败");
                }
            }
        }

        //前置增强
        @Before(value = "execution(* com.ssmbook2020..*.*(..))")
        public void methodBefore(JoinPoint join){
            String method = join.getSignature().getName();
            Object[] args = join.getArgs();
            if ("login".equals(method)){
                logger.info(new Timestamp(System.currentTimeMillis()) + " " +
args[0] + "开始登录");
            }
        }
    }
```

（3）配置 applicationContext.xml 文件，在 XML 文件头部添加 AOP 命名空间，以使用与 AOP 相关的标签。同时因为代码中用到了 p 的方式注解，所以也添加了 p 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:p="http://www.springframework.org/schema/p"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.1.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-4.1.xsd">
        <!-- 日志记录类 -->
        <bean id="loggerAdvice" class="com.ssmbook2020.utils.LoggerAdvice" />
        <!-- 业务类 -->
        <bean id="userService" class="com.ssmbook2020.service.impl.
UserServiceImpl" />
        <!-- 织入使用注解定义的增强，需要引入 AOP 命名空间 -->
        <aop:aspectj-autoproxy />
    </beans>
```

上面的配置将所有的 JavaBean 加入 Spring 容器中，其中最重要的是<aop:aspectj-autoproxy />，表示所有的 AOP 自动代理，通过注解的方式织入。

（4）构建测试类，其代码如下。

```
package com.ssmbook2020.test;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.ssmbook2020.service.UserService;
public class TestAop {
   public static void main(String[] args){
      ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
      //得到业务类
      UserService userService = (UserService) context.getBean("userService");
      //运行业务登录方法
      userService.login("newboy", "520");
      context.close();
   }
}
```

测试类代码的运行结果如下。

```
INFO - 2021-01-20 12:48:22.617 newboy 开始登录
业务方法 login 运行，正在登录...
INFO - 2021-01-20 12:48:22.618 newboy 登录成功
```

如果把 userService.login("newboy", "520")换成 userService.login("Lina", "1314")，则运行结果如下。

```
INFO - 2021-01-20 12:48:05.503 Lina 开始登录
业务方法 login 运行，正在登录...
INFO - 2021-01-20 12:48:05.504 Lina 登录失败
```

在业务类的代码运行时，该方法的前后各输出了日志的内容，这就是代码的织入，也称为前置或后置增强。

通过使用注解，使得 Spring AOP 的配置被极大地简化。如果把业务类和切面类在 Spring IoC 中的配置也改成注解，则可以进一步简化 XML 的配置内容。

3.4 本章小结

本章介绍了 Spring 的另一个重要特性 AOP，它是 Spring 框架最核心、最基础的技术之一。Spring AOP 的实现原理是使用了动态代理模式，在此重点介绍了 JDK 的代理模式。

本章分别讲解了 XML 配置和注解两种方式实现 Spring AOP，注解的方式相对代码量更少，后期使用比较多。对 Spring AOP 的学习只是 Spring 框架学习的开始，它是一个庞大的框架，其目标是让一切 Java EE 的开发都变得更简洁。

习 题 3

1. 什么是 Spring AOP？
2. Spring AOP 的实现原理是什么？
3. 分别使用 XML 配置和注解方式实现本章案例。