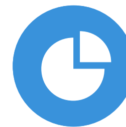
# 第3章  Spring AOP

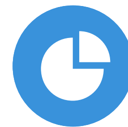
湖南科技大学
计算机科学与工程学院

# 学习目标/Target



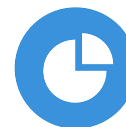
- 了解Spring AOP的概念及其术语
- 熟悉Spring AOP的JDK动态代理
- 熟悉Spring AOP的CGLib动态代理
- 掌握基于XML的AOP实现
- 掌握基于注解的AOP实现

# 章节概述/ Summary



Spring的AOP模块是Spring框架体系中十分重要的内容，该模块一般适用于具有横切逻辑的场景，如访问控制、事务管理和性能监控等，本章将对Spring AOP的相关知识进行详细讲解。

# 目录/Contents

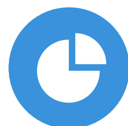


01 Spring AOP介绍

02 Spring AOP的实现机制

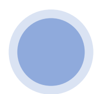
03 基于XML的AOP实现

04 基于注解的AOP实现

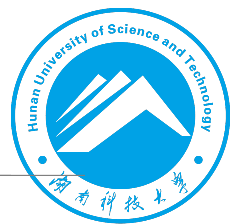


# 3.1 Spring AOP介绍

# 3.1.1 Spring AOP概述



了解Spring AOP概述，能够说出什么是Spring AOP

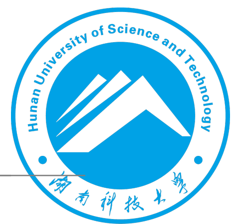



## AOP概述

AOP的全称是Aspect Oriented Programming，即面向切面编程。和OOP不同，AOP主张将程序中相同的业务逻辑进行横向隔离，并将重复的业务逻辑抽取到一个独立的模块中，以达到提高程序可重用性和开发效率的目的。

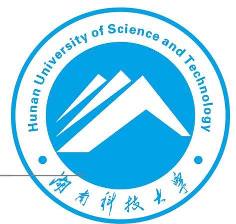
在传统的业务处理代码中，通常都会进行事务处理、日志记录等操作。虽然使用OOP可以通过组合或者继承的方式来达到代码的重用，但如果要实现某个功能（如日志记录），同样的代码仍然会分散到各个方法中。

# 3.1.1 Spring AOP概述




## 未使用AOP的面向切面编程案例

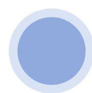
例如，订单系统中有添加订单信息、更新订单信息和删除订单信息3个方法，这3个方法中都包含事务管理业务代码，订单系统的逻辑如图所示。




## AOP面向切面编程的优势

由订单系统可知，添加订单信息、修改订单信息、删除订单信息的方法体中都包含事务管理的业务逻辑，这就带来了一定数量的重复代码并使程序的维护成本增加。基于AOP的面向切面编程，可以为此类问题提供解决方案，AOP可以将事务管理的业务逻辑从这三个方法体中抽取到一个可重用的模块，进而降低横向业务逻辑之间的耦合，减少重复代码。AOP的使用，使开发人员在编写业务逻辑时可以专心于核心业务，而不用过多地关注其他业务逻辑的实现，不但提高了开发效率，而且增强了代码的可维护性。

# 3.1.2 Spring AOP术语



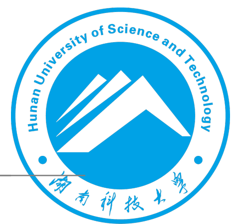
了解Spring AOP术语，能够说出Spring AOP的常用术语及术语的意思

# 3.1.2 Spring AOP术语

## AOP术语

AOP并不是一个新的概念，AOP中涉及很多术语，如切面、连接点、切入点、通知/增强处理、目标对象、织入、代理和引介等，下面针对AOP的常用术语进行简单介绍。

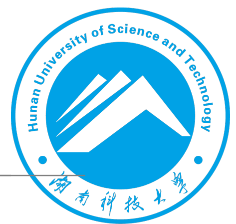
# 3.1.2 Spring AOP术语




## 切面（Aspect）

切面是指关注点形成的类（关注点是指类中重复的代码），通常是指封装的、用于横向插入系统的功能类（如事务管理、日志记录等）。在实际开发中，该类被Spring容器识别为切面，需要在配置文件中通过<bean>元素指定。

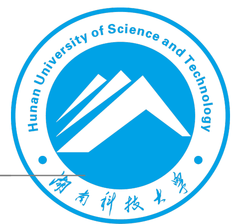
## 3.1.2 Spring AOP术语




### 连接点（Joinpoint）

连接点是程序执行过程中某个特定的节点，例如，某方法调用时或处理异常时。在Spring AOP中，一个连接点通常是一个方法的执行。

# 3.1.2 Spring AOP术语




## 切入点（Pointcut）

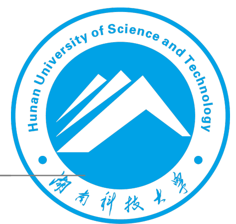
当某个连接点满足预先指定的条件时，AOP就能够定位到这个连接点，在连接点处插入切面，该连接点也就变成了切入点。

# 3.1.2 Spring AOP术语

## 通知/增强处理（Advice）

通知/增强处理就是插入的切面程序代码。可以将通知/增强处理理解为切面中的方法，它是切面的具体实现。

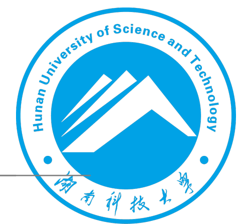
# 3.1.2 Spring AOP术语



## 目标对象（Target）

目标对象是指被插入切面的方法，即包含主业务逻辑的类对象。或者说是被一个或者多个切面所通知的对象。

## 3.1.2 Spring AOP术语




### 织入（Weaving）

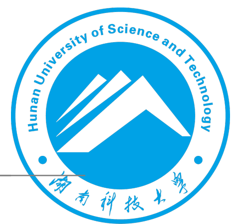
将切面代码插入到目标对象上，从而生成代理对象的过程。织入可以在编译时，类加载时和运行时完成。**在编译时进行织入就是静态代理，而在运行时进行织入则是动态代理。**

# 3.1.2 Spring AOP术语

## 代理（Proxy）

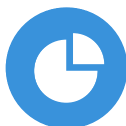
将通知应用到目标对象之后，程序动态创建的通知对象，就称为代理。代理类既可能是和原类具有相同接口的类，也可能是原类的子类，可以采用调用原类相同的方式调用代理类。

# 3.1.2 Spring AOP术语



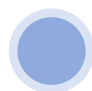

## 引介（Introduction）

引介是一种特殊的通知，它可为目标对象添加一些属性和方法。这样，即使一个业务类原本没有实现某一个接口，通过AOP的引介功能，也可以动态地为该业务类添加接口的实现逻辑，让业务类成为这个接口的实现类。



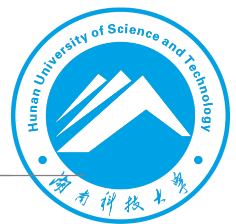
# 3.2 Spring AOP的实现机制

# 3.2.1 JDK动态代理



熟悉JDK动态代理，能够灵活应用JDK动态代理的机制

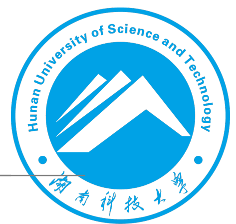
# 3.2.1 JDK动态代理




## Spring AOP的默认代理方式

默认情况下，Spring AOP使用JDK动态代理，JDK动态代理是通过java.lang.reflect.Proxy 类实现的，可以调用Proxy类的newProxyInstance()方法创建代理对象。JDK动态代理可以实现无侵入式的代码扩展，并且可以在不修改源代码的情况下，增强某些方法。

## 3.2.1 JDK动态代理




接下来，通过一个案例演示Spring中JDK动态代理的实现过程，案例具体实现步骤如下。

**STEP 01**

在IDEA中创建一个名为chapter08的Maven项目，然后在项目的pom.xml文件中加载需使用到的Spring基础包和Spring的依赖包。

# 3.2.1 JDK动态代理

**STEP 02**

创建接口UserDao，在UserDao接口中编写添加和删除的方法。

```
package com.itheima.demo01;
public interface UserDao {
    public void addUser();
    public void deleteUser();
}
```

# 3.2.1 JDK动态代理

**STEP 03**

创建UserDao接口的实现类UserDaoImpl，分别实现接口中的方法。

```
package com.itheima.demo01;
public class UserDaoImpl implements UserDao {
        public void addUser() {
                System.out.println("添加用户");      }
        public void deleteUser() {
                System.out.println("删除用户");      }
}
```

# 3.2.1 JDK动态代理

**STEP 04**

创建切面类MyAspect，在该类中定义一个模拟权限检查的方法和一个模拟日志记录的方法，这两个方法就是切面中的通知。

```
package com.itheima.demo01;
// 切面类：存在多个通知Advice（增强的方法）
public class MyAspect {
    public void check_Permissions(){
        System.out.println("模拟检查权限...");        }
    public void log(){
        System.out.println("模拟记录日志...");        }
}
```

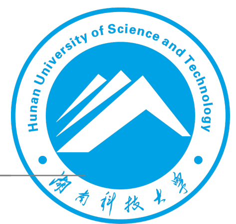
# 3.2.1 JDK动态代理

**STEP 05**

创建代理类MyProxy，该类需要实现InvocationHandler接口设置代理类的调用处理程序。在代理类中，通过newProxyInstance()生成代理方法。

```
public class MyProxy implements InvocationHandler {
    private UserDao userDao;
    public  Object createProxy(UserDao userDao) {
        this.userDao = userDao;
        ClassLoader classLoader = MyProxy.class.getClassLoader(); // 1.类加载器
        Class[] classes = userDao.getClass().getInterfaces(); // 2.被代理对象实现的所有接口
        return  Proxy.newProxyInstance(classLoader,classes,this); // 3.返回代理对象
    }
    // 所有动态代理类的方法调用，都会交由invoke()方法去处理。篇幅问题这里省略invoke()方法
```

# 3.2.1 JDK动态代理



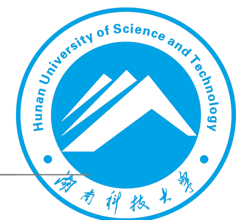

## newProxyInstance()方法的3个参数

第1个参数是classLoader，表示当前类的类加载器。

第2个参数是classes，表示被代理对象实现的所有接口。

第3个参数是this，表示代理类JdkProxy本身。

# 3.2.1 JDK动态代理




**STEP 06**

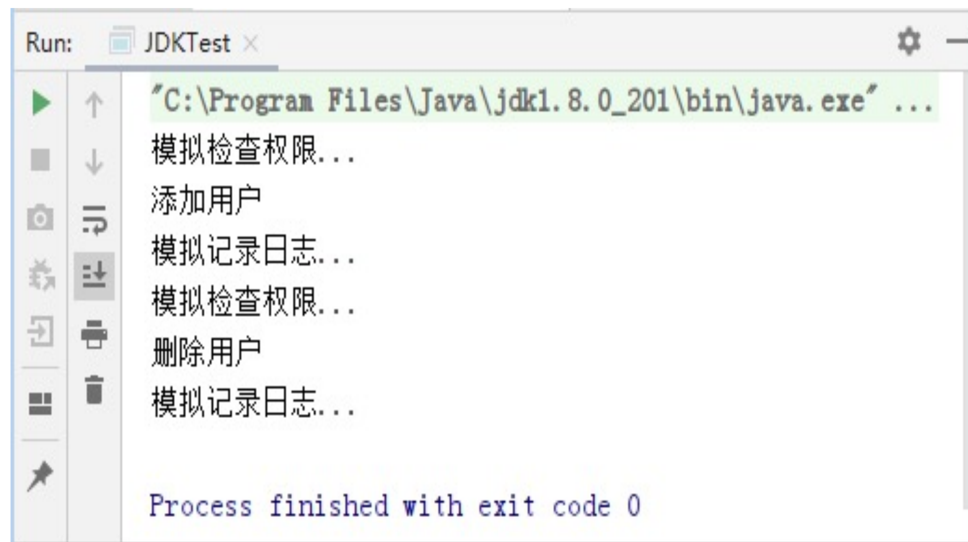
创建测试类JDKTest。在该类中的main()方法中创建代理对象jdkProxy和目标对象userDao，然后从代理对象jdkProxy中获得对目标对象userDao增强后的对象userDao1，最后调用userDao1对象中的添加和删除方法。

```
public class JDKTest {
    public static void main(String[] args) {
        MyProxy jdkProxy = new MyProxy();// 创建代理对象
        UserDao userDao = new UserDaoImpl();// 创建目标对象
        // 从代理对象中获取增强后的目标对象
        UserDao userDao1 = (UserDao) jdkProxy.createProxy(userDao);
        // 执行方法
        userDao1.addUser();
        userDao1.deleteUser();
}}
```

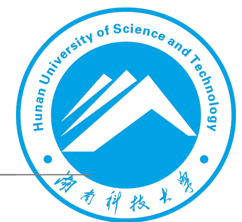
# 3.2.1 JDK动态代理

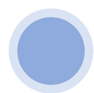
**STEP 07**

在IDEA中启动JDKTest类，控制台会输出结果。



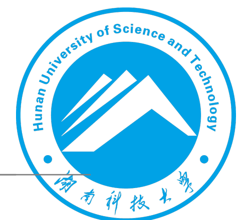
# 3.2.2 CGLib代理





熟悉CGLib动态代理，能够灵活应用CGLib动态代理的机制

## 3.2.2 CGLib动态代理




### JDK与CGLib动态代理的比较

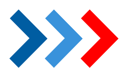
JDK动态代理存在缺陷，它只能为接口创建代理对象，当需要为类创建代理对象时，就需要使用CGLib（Code Generation Library）动态代理，CGLib动态代理不要求目标类实现接口，它采用底层的字节码技术，通过继承的方式动态创建代理对象。Spring的核心包已经集成了CGLib所需要的包，所以开发中不需要另外导入JAR包。

# 3.2.2 CGLib动态代理

接下来通过一个案例演示CGLib动态代理的实现过程，具体步骤如下。

**STEP 01** 创建目标类UserDao，在该类中编写添加用户和删除用户的方法。

```
package com.itheima.demo02;
public class UserDao {
    public void addUser(){
        System.out.println("添加用户");        }
    public void deleteUser(){
        System.out.println("删除用户");        }
}
```



# 3.2.2 CGLib动态代理

STEP 02

创建代理类CglibProxy，该代理类需要实现MethodInterceptor接口用于设置代理类的调用处理程序，并实现接口中的intercept()方法。

```
public class CglibProxy implements MethodInterceptor {
    public  Object createProxy(Object target) {// 代理方法
        Enhancer enhancer = new Enhancer();// 创建一个动态类对象
        enhancer.setSuperclass(target.getClass());// 确定需要增强的类，设置其父类
        enhancer.setCallback(this);// 添加回调函数
        return enhancer.create();// 返回创建的代理类
    }
    // intercept()方法省略
}
```

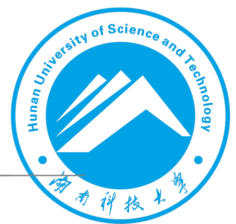
# 3.2.2 CGLib动态代理

**STEP 03**

创建测试类CglibTest，在main()方法中首先创建代理对象cglibProxy和目标对象userDao，然后从代理对象cglibProxy中获得增强后的目标对象userDao1，最后调用userDao1对象的添加和删除方法。

```
public class CglibTest {
    public static void main(String[] args) {
        CglibProxy cglibProxy = new CglibProxy(); // 创建代理对象
        UserDao userDao = new UserDao(); // 创建目标对象
        // 获取增强后的目标对象
        UserDao userDao1 = (UserDao)cglibProxy.createProxy(userDao);
        // 执行方法
        userDao1.addUser();
        userDao1.deleteUser();
}}
```

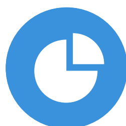
# 3.2.2 CGLib动态代理




**STEP 04**

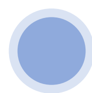
在IDEA中启动CglibTest类，控制台会输出结果。

```
Run:  CglibTest
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
模拟检查权限...
添加用户
模拟记录日志...
模拟检查权限...
删除用户
模拟记录日志...

Process finished with exit code 0
```

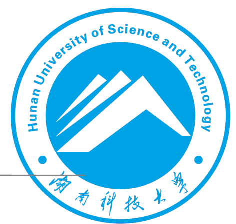


# 3.3 基于XML的AOP实现

# 3.3 基于XML的AOP实现



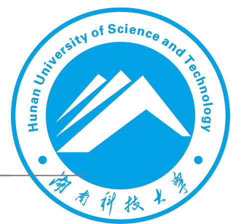
掌握基于XML的AOP实现，能够基于XML的方式实现Spring AOP




## 使用AOP代理对象的好处

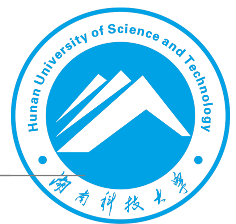
因为Spring AOP中的代理对象由IoC容器自动生成，所以开发者无须过多关注代理对象生成的过程，只需选择连接点、创建切面、定义切点并在XML文件中添加配置信息即可。Spring提供了一系列配置Spring AOP的XML元素。

# 3.3 基于XMl的AOP实现




## 配置Spring AOP的XML元素

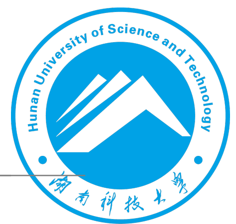
| 元素 | 描述 |
| --- | --- |
| <aop:config> | Spring AOP配置的根元素 |
| <aop:aspect> | 配置切面 |
| <aop:advisor> | 配置通知器 |
| <aop:pointcut> | 配置切点 |
| <aop:before> | 配置前置通知,在目标方法执行前实施增强,可以应用于权限管理等功能 |
| <aop:after> | 配置后置通知,在目标方法执行后实施增强,可以应用于关闭流、上传文件、删除临时文件等功能 |
| <aop:around> | 配置环绕方式,在目标方法执行前后实施增强,可以应用于日志、事务管理等功能 |
| <aop:after-returning> | 配置返回通知,在目标方法成功执行之后调用通知 |
| <aop:after-throwing> | 配置异常通知,在方法抛出异常后实施增强,可以应用于处理异常记录日志等功能 |




## 配置切面

在Spring的配置文件中，配置切面使用的是<aop:aspect>元素，该元素会将一个已定义好的Spring Bean转换成切面Bean，因此，在使用<aop:aspect>元素之前，要在配置文件中先定义一个普通的Spring Bean。Spring Bean定义完成后，通过<aop:aspect>元素的ref属性即可引用该Bean。配置<aop:aspect>元素时，通常会指定id和ref两个属性。

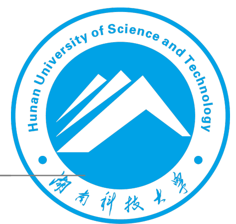
# 3.3 基于XML的AOP实现




## <aop:aspect>元素的id属性和ref属性的描述

| 属性名称 | 描述 |
| --- | --- |
| id | 用于定义该切面的唯一标识 |
| ref | 用于引用普通的Spring Bean |

# 3.3 基于XMl的AOP实现




## 配置切入点

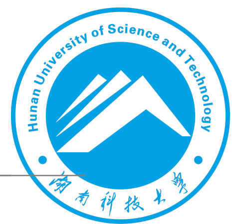
在Spring的配置文件中，切入点是通过<aop:pointcut>元素来定义的。当<aop:pointcut>元素作为<aop:config>元素的子元素定义时，表示该切入点是全局的，它可被多个切面共享；当<aop:pointcut>元素作为<aop:aspect>元素的子元素时，表示该切入点只对当前切面有效。定义<aop:pointcut>元素时，通常会指定id、expression属性。

# 3.3 基于XML的AOP实现

## <aop:pointcut>元素的id属性和expression属性描述

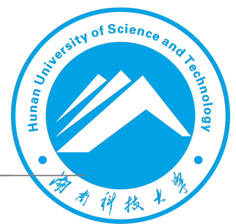
| 属性名称 | 描述 |
| --- | --- |
| id | 用于指定切入点的唯一标识 |
| expression | 用于指定切入点关联的切入点表达式 |

# 3.3 基于XML的AOP实现




## Spring AOP切入点表达式的基本格式

```
execution(modifiers-pattern?ret-type-pattern
declaring-type-pattern?
name-pattern(param-pattern) throws-pattern?)
```




## execution表达式各部分参数说明

modifiers-pattern：表示定义的目标方法的访问修饰符，如public、private等。

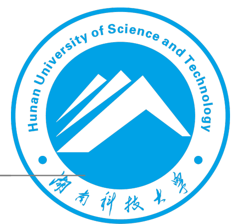
ret-type-pattern：表示定义的目标方法的返回值类型，如void、String等。

declaring-type-pattern：表示定义的目标方法的类路径，如com.itheima.jdk.UserDaoImpl。

name-pattern：表示具体需要被代理的目标方法，如add()方法。

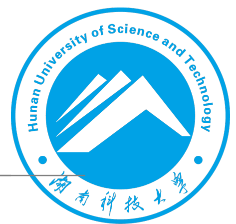
param-pattern：表示需要被代理的目标方法包含的参数，本章示例中目标方法参数都为空。

throws-pattern：表示需要被代理的目标方法抛出的异常类型。




## 配置通知

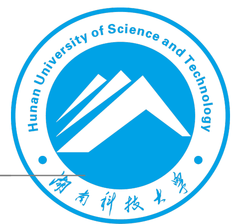
在Spring的配置文件中，使用<aop:aspect>元素配置了5种常用通知，分别为前置通知、后置通知、环绕通知、返回通知和异常通知。




## <aop:aspect>元素的常用属性

| 属性 | 描述 |
| --- | --- |
| pointcut | 该属性用于指定一个切入点表达式,Spring将在匹配该表达式的连接点时织入该通知。 |
| pointcut-ref | 该属性指定一个已经存在的切入点名称,如配置代码中的myPointCut。通常pointcut和pointcut-ref两个属性只需要使用其中一个即可。 |
| method | 该属性指定一个方法名,指定将切面Bean中的该方法转换为增强处理。 |
| throwing | 该属性只对<after-throwing>元素有效,它用于指定一个形参名,异常通知方法可以通过该形参访问目标方法所抛出的异常。 |
| returning | 该属性只对<after-returning>元素有效,它用于指定一个形参名,后置通知方法可以通过该形参访问目标方法的返回值。 |

# 3.3 基于XML的AOP实现




接下来通过一个案例演示如何在Spring中使用XML实现Spring AOP，具体实现步骤如下。

**STEP 01**

在chapter08项目的pom.xml文件中导入AspectJ框架的相关JAR包。

```
<!-- aspectjrt包的依赖 -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.9.1</version>
</dependency>
<!-- aspectjweaver包的依赖 -->
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.6</version>
</dependency>
```

# 3.3 基于XML的AOP实现

**STEP 02**

创建接口UserDao，并在该接口中编写添加、删除、修改和查询的方法。

```
package com.itheima.demo03;
public interface UserDao {
    public void insert();
    public void delete();
    public void update();
    public void select();
}
```

# 3.3 基于XML的AOP实现

**STEP 03**

创建UserDao接口的实现类UserDaoImpl，实现UserDao接口中的方法。

```
public class UserDaoImpl implements UserDao{
    public void insert() {
        System.out.println("添加用户信息"); }
    public void delete() {
        System.out.println("删除用户信息"); }
    public void update() {
        System.out.println("更新用户信息"); }
    public void select() {
        System.out.println("查询用户信息"); }
}
```

# 3.3 基于XML的AOP实现

**STEP 04**

创建XmlAdvice类，用于定义通知。

```
public class XmlAdvice {
    // 前置通知
    public void before(JoinPoint joinPoint){
        System.out.print("这是前置通知!");
        System.out.print("目标类是："+joinPoint.getTarget());
        System.out.println("，被织入增强处理的目标方法为："+
                      joinPoint.getSignature().getName());
    }
    // 因为篇幅问题，其他通知省略：返回通知、环绕通知、异常通知、后置通知
}
```

# 3.3 基于XML的AOP实现

**STEP 05**

创建applicationContext.xml文件，在该文件中引入AOP命名空间，使用<bean>元素添加Spring AOP的配置信息。

```
<！-- 注册bean省略，下面内容为配置Spring AOP-->
<aop:config>
    <aop:pointcut id="pointcut" expression="execution(*
        com.itheima.demo03.UserDaoImpl.*(..))"/><!-- 指定切点 -->
    <aop:aspect ref ="xmlAdvice"><!-- 指定切面 -->
        <aop:before method="before" pointcut-ref="pointcut"/><!-- 指定前置通知 -->
        <aop:after-returning method="afterReturning" pointcut-ref="pointcut"/>
        <aop:around method="around" pointcut-ref="pointcut"/>-- 指定环绕方式 -->
        <aop:after-throwing method="afterException" pointcut-ref="pointcut"/>
        <aop:after method="after" pointcut-ref="pointcut"/><!-- 指定后置通知 -->
    </aop:aspect>
</aop:config>
```

# 3.3 基于XML的AOP实现

**STEP 06**

创建测试类TestXml，测试基于XML的AOP实现。

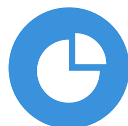
```
public class TestXml{
    public static void main(String[] args){
        ApplicationContext context=new  ClassPathXmlApplicationContext("applicationContext.xml");
        UserDao userDao=context.getBean("userDao",UserDao.class);
        userDao.delete();
        userDao.insert();
        userDao.select();
        userDao.update();
    }
}
```

# 3.3 基于XML的AOP实现

**STEP 07**

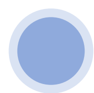
在IDEA中启动TestXml类，控制台会输出结果。

```
Run: TestXml
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
这是前置通知!目标类是：com.itheima.demo03.UserDaoImpl@7692d9cc，被织入增强处理的目标方法为：delete
这是环绕通知之前的部分！
删除用户信息
这是后置通知！
这是环绕通知之后的部分！
这是返回通知（方法不出现异常时调用）！被织入增强处理的目标方法为：delete

这是前置通知!目标类是：com.itheima.demo03.UserDaoImpl@7692d9cc，被织入增强处理的目标方法为：insert
这是环绕通知之前的部分！
添加用户信息
这是后置通知！
这是环绕通知之后的部分！
这是返回通知（方法不出现异常时调用）！被织入增强处理的目标方法为：insert

这是前置通知!目标类是：com.itheima.demo03.UserDaoImpl@7692d9cc，被织入增强处理的目标方法为：select
这是环绕通知之前的部分！
查询用户信息
这是后置通知！
这是环绕通知之后的部分！
这是返回通知（方法不出现异常时调用）！被织入增强处理的目标方法为：select

这是前置通知!目标类是：com.itheima.demo03.UserDaoImpl@7692d9cc，被织入增强处理的目标方法为：update
这是环绕通知之前的部分！
更新用户信息
这是后置通知！
这是环绕通知之后的部分！
这是返回通知（方法不出现异常时调用）！被织入增强处理的目标方法为：update

Process finished with exit code 0
```



# 3.4 基于注解的AOP实现

# 3.4 基于注解的AOP实现



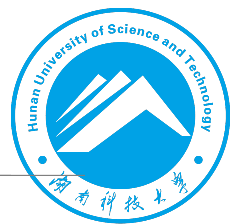
掌握基于注解的AOP实现，能够在程序中熟练运用注解的方式实现Spring AOP

# 3.4 基于注解的AOP实现

## Spring AOP的注解

| 元素 | 描述 |
| --- | --- |
| @Aspect | 配置切面 |
| @Pointcut | 配置切点 |
| @Before | 配置前置通知 |
| @After | 配置后置通知 |
| @Around | 配置环绕方式 |
| @AfterReturning | 配置返回通知 |
| @AfterThrowing | 配置异常通知 |

# 3.4 基于注解的AOP实现



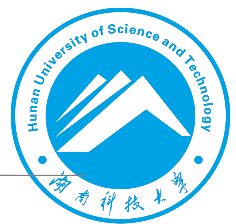

下面通过一个案例演示基于注解的AOP的实现，案例具体实现步骤如下。

**STEP 01**

创建AnnoAdvice类，用于定义通知。

```
@Aspect
public class AnnoAdvice {/*
    @Pointcut("execution( * com.itheima.demo03.UserDaoImpl.*(..))")
    @Before("poincut()")
    @AfterReturning("poincut()")
    @Around("poincut()")
    @AfterThrowing("poincut()")
    @After( “poincut()")
     使用以上注解分别定义切点、前置通知、返回通知、环绕通知、异常通知、后置通知*/
}
```

# 3.4 基于注解的AOP实现




**STEP 02**

创建applicationContext-Anno.xml文件，在该文件中引入AOP命名空间，使用<bean>元素添加Spring AOP的配置信息。

```
<!-- 注册Bean -->
<bean name="userDao" class="com.itheima.demo03.UserDaoImpl"/>
<bean name="AnnoAdvice" class="com.itheima.demo04.AnnoAdvice"/>
<!-- 开启@aspectj的自动代理支持 -->
<aop:aspectj-autoproxy/>
```

**STEP 03**

创建测试类TestAnnotation，用于测试基于注解的AOP实现。

```
public class TestAnnotation {
    public static void main(String[] args){
        ApplicationContext context = new
        ClassPathXmlApplicationContext("applicationContext-Anno.xml");
        UserDao userDao = context.getBean("userDao",UserDao.class);
        userDao.delete();
        userDao.insert();
        userDao.select();
        userDao.update();
}}
```

# 3.4 基于注解的AOP实现

**STEP 04**

在IDEA中启动TestAnnotation类，控制台会输出结果。

Run: TestXml

```
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
这是前置通知!目标类是：com.itheima.demo03.UserDaoImpl@7692d9cc，被织入增强处理的目标方法为：delete
这是环绕通知之前的部分!
删除用户信息
这是后置通知!
这是环绕通知之后的部分!
这是返回通知（方法不出现异常时调用）!被织入增强处理的目标方法为：delete

这是前置通知!目标类是：com.itheima.demo03.UserDaoImpl@7692d9cc，被织入增强处理的目标方法为：insert
这是环绕通知之前的部分!
添加用户信息
这是后置通知!
这是环绕通知之后的部分!
这是返回通知（方法不出现异常时调用）!被织入增强处理的目标方法为：insert

这是前置通知!目标类是：com.itheima.demo03.UserDaoImpl@7692d9cc，被织入增强处理的目标方法为：select
这是环绕通知之前的部分!
查询用户信息
这是后置通知!
这是环绕通知之后的部分!
这是返回通知（方法不出现异常时调用）!被织入增强处理的目标方法为：select

这是前置通知!目标类是：com.itheima.demo03.UserDaoImpl@7692d9cc，被织入增强处理的目标方法为：update
这是环绕通知之前的部分!
更新用户信息
这是后置通知!
这是环绕通知之后的部分!
这是返回通知（方法不出现异常时调用）!被织入增强处理的目标方法为：update

Process finished with exit code 0
```

# 本章小结

本章主要讲解了Spring中的AOP。首先介绍了Spring AOP，包括Spring AOP的概述和Spring AOP的术语；然后讲解了Spring AOP的实现机制，包括JDK动态代理和CGLib动态代理；接着讲解了基于XML的AOP实现，并使用案例的方式实现了基于XML文件的AOP；最后讲解了基于注解的AOP实现。通过本章的学习，读者可以对Spring AOP有基础的了解，为框架开发奠定基础。