# 课程目标

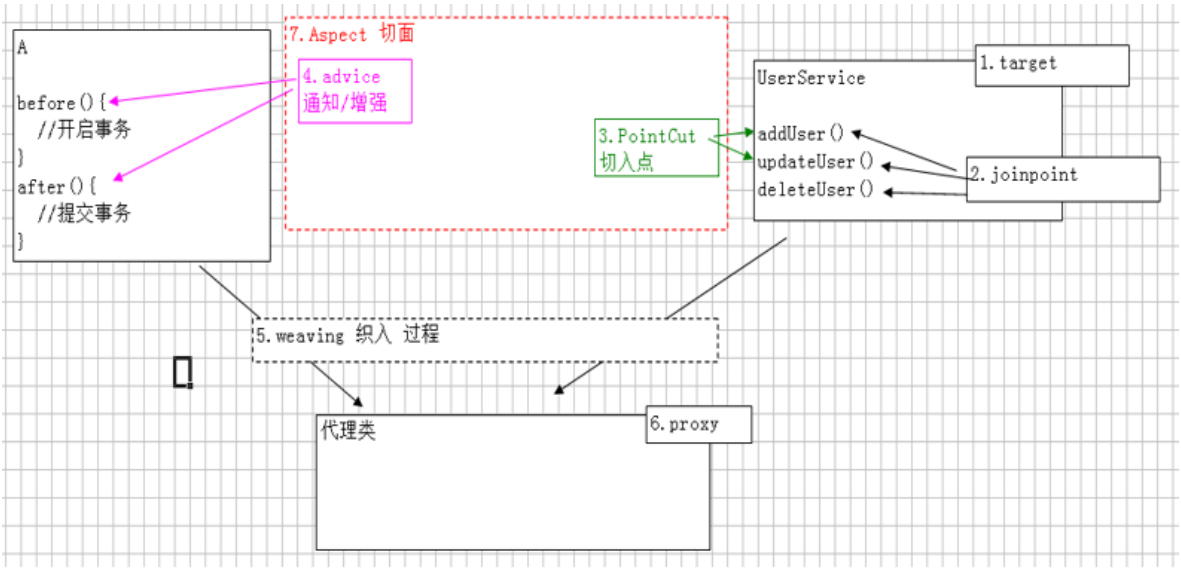
# 1. AOP 介绍

## 1.1 AOP 简介

- `AOP` 是 `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` （面向对象编程）的补充，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。
- 经典应用：**事务管理、性能监视、安全检查、缓存、日志**等

## 1.2 AOP 术语

1. `target` ：目标类，需要被代理的类 HotelService。
2. `joinpoint` ：连接点：是指那些可能被拦截到的方法 deleteById update() insert() 。
3. `pointCut` ： 切入点：已经被增强的连接点 deleteById。
4. `advice` ：通知/增强，增强代码。
5. **`weaving` ：织入是指把增强 `advice` 应用到目标对象 `target` 来创建新的代理对象 `proxy` 的过程.**
6. `proxy` ：代理类
7. `Aspect` :切面：是切入点 `pointcut` 和通知/增强 `advice` 的结合。



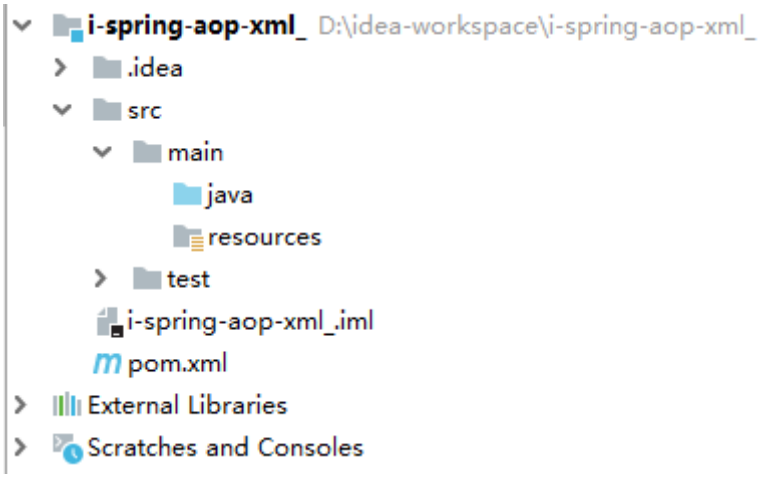

# 2. Spring AOP

- `Spring AOP` 使用纯Java实现，不需要专门的编译过程和类加载器，**在运行期通过动态代理方式向目标类织入增强代码。**(JDK动态代理 CGLIB)

- `AspectJ` 是一个基于Java语言的**AOP框架**，`Spring2.0` 开始，`Spring AOP` 引入对 `Aspect` 的支持。
- `@AspectJ` 是 `AspectJ 1.5` 新增功能，通过 `JDK5` 注解技术，允许直接在Bean类中定义切面，
- 新版本Spring2.0 框架，建议使用 `AspectJ` 方式来开发 `AOP`

# 3. 快速入门

## 3.1 创建Maven工程

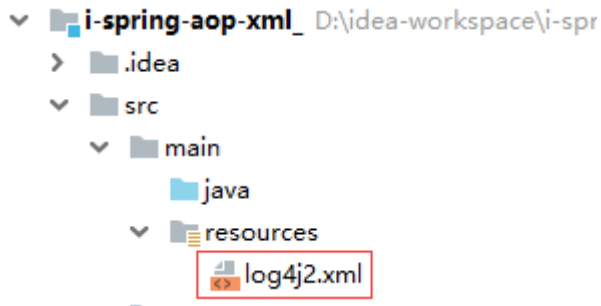



## 3.2 项目依赖

```
<!--单元测试开始-->
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>5.1.10.RELEASE</version>
    <scope>test</scope>
</dependency>
<!--单元测试结束-->

<!--日志 log4j2-->
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.12.1</version>
</dependency>

<!--spring -->
<dependency>
    <groupId>org.springframework</groupId>
```

```
    <artifactId>spring-context</artifactId>
    <version>5.1.10.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
    <version>5.1.10.RELEASE</version>
</dependency>
```

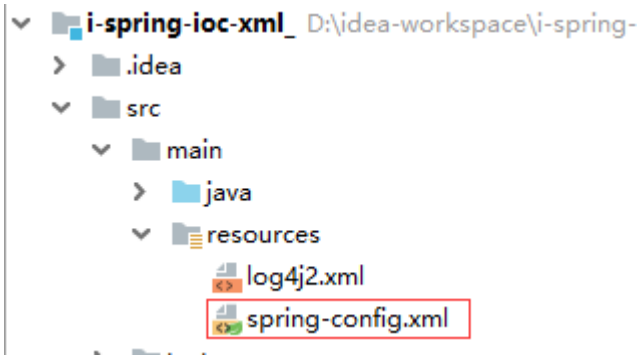
## 3.3 日志配置




```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <!--先定义所有的appender-->
    <appenders>
        <!--输出控制台的配置-->
        <console name="Console" target="SYSTEM_OUT">
            <!--输出日志的格式-->
            <!--<patternlayout pattern="%d{yyyy-MM-dd HH:mm:ss} [%p] %m %c %n"/>-->
            <!--<patternlayout pattern="[%p] %m %n"/>-->
            <!--<PatternLayout
                    pattern="%style{%date{DEFAULT}}{yellow} %highlight{%-5level}{FATAL=bg_red, ERROR=red, WARN=yellow,INFO=green} %m%n"
                    disableAnsi="false" noConsoleNoAnsi="false"/>-->
            <PatternLayout
                    pattern="%highlight{[%p]}{ERROR=red, WARN=yellow,INFO=black,DEBUG=green} %style{%F [%L]}{blue} - %highlight{%m}{ERROR=red, WARN=yellow,INFO=black,DEBUG=green}%n"
                    disableAnsi="false" noConsoleNoAnsi="false"/>

        </console>
    </appenders>

    <!-- 然后定义logger，只有定义了logger并引入的appender，appender才会生效-->
    <!-- 日志级别以及优先级排序: OFF > FATAL > ERROR > WARN > INFO > DEBUG > TRACE > ALL -->
    <loggers>
        <root level="DEBUG">
            <!--输出到控制台-->
            <appender-ref ref="Console"/>
        </root>
        <!--org.springframework-->
        <logger name="org.springframework" level="ERROR"/>
    </loggers>
```

```
30    </configuration>
```

## 3.4 Spring配置文件




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

</beans>
```

## 3.5 目标类（target）

```java
package com.imcode.spring.service.UserService;
public class UserService {

    private static Logger logger = LogManager.getLogger(UserService.class);

    public User login(String username, String password) throws Exception {
        if ("admin".equals(username) && "123456".equals(password)) {
            User user = new User(username,password);
            //logger.info("用户{}登录系统成功", username);
            return user;
        } else {
            //logger.error("用户{}登录系统失败,用户名或密码错误", username);
            throw new Exception("用户名或密码错误");
        }
    }
}
```

## 3.6 通知/增强(advice)

**通知类型**

- @Before: 前置通知
  - 在方法执行之前执行
  - 如果通知抛出异常，阻止方法运行
- @After: 后置通知

    - 在方法执行之后执行
    - 无论方法中是否出现异常该通知都执行
- @AfterRunning: 返回通知
    - 在方法返回结果之后执行
    - 如果方法中抛出异常，该通知不执行
- @AfterThrowing: 异常通知
    - 方法抛出异常后执行，
    - 如果方法没有抛出异常，该通知不执行
- @Around：环绕通知
    - 方法执行前后分别执行，可以阻止方法的执行
    - **必须手动执行目标方法**

```java
package com.imcode.spring.aspect;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
/**
 * 增强/通知 (advice)
 */
public class LogAspect {
    private static Logger logger = LogManager.getLogger(LogAspect.class);
    /**
     * 前置通知
     * - 在方法执行之前执行
     * - 如果通知抛出异常，阻止方法运行
     * @param joinPoint
     */
    public void before(JoinPoint joinPoint){
        logger.info("@Before 前置通知:{}",joinPoint.getSignature().getName() );
    }

    /**
     * 返回通知
     * - 在目标方法返回结果之后执行
     * - 如果方法中抛出异常，该通知不执行
     * - 可以获取到目标方法的返回值
     * @param joinPoint  连接点 存储拦截到的方法的相关信息
     * @param result  目标方法执行后的返回值
     */
    public void afterReturning(JoinPoint joinPoint, Object result){
        // 目标方法的名称
        String methodName = joinPoint.getSignature().getName();
        // 目标方法的参数
        Object[] params = joinPoint.getArgs();
        // 目标类的对象
        Object target = joinPoint.getTarget();

//        System.out.println("目标方法的名称:" + methodName);
//        System.out.println("目标方法的参数:" + Arrays.toString(params));
//        System.out.println("目标类的对象:" + target);
//        System.out.println("方法返回值:" + result);
        logger.info("@AfterReturning 返回通知: 用户 {} 登录系统成功...",params[0]);
    }
```

```
44
45      /**
46       * 后置通知
47       * - 在方法执行之后执行
48       * - 无论方法中是否出现异常该通知都执行
49       * @param joinPoint
50       */
51      public void after(JoinPoint joinPoint) {
52          logger.info("@After 后置通知:{}",joinPoint.getSignature().getName() );
53      }
54
55      /**
56       * 异常通知
57       * - 方法抛出异常后执行，
58       * - 如果方法没有抛出异常，该通知不执行
59       * @param joinPoint
60       * @param e
61       */
62      public void afterThrowing(JoinPoint joinPoint, Throwable e) {
63          // 目标方法的参数
64          Object[] params = joinPoint.getArgs();
65          logger.error("@afterThrowing 异常通知: 用户 {} 登录系统异常:{}",
66                  params[0], e.getMessage());
67      }
68      /**
69       * 环绕通知
70       * - 方法执行前后分别执行，可以阻止方法的执行
71       * - 必须手动执行目标方法
72       * @param joinPoint
73       */
74      public Object around(ProceedingJoinPoint joinPoint){
75          logger.info("环绕通知（@Before 前置通知）:");
76          Object obj = null;
77          try {
78              // 手动执行目标方法
79              // obj = joinPoint.proceed();
80              Object[] params = joinPoint.getArgs();
81              params[1] = "111111";
82              // 在运行期动态的修改传入方法参数的值
83              obj = joinPoint.proceed(params);
84              logger.info("环绕通知（@AfterRunning 返回通知）:");
85          } catch (Throwable e) {
86              logger.info("环绕通知（@AfterThrowing 异常通知）:");
87              e.printStackTrace();
88          } finally {
89              logger.info("环绕通知（@After 后置通知）:");
90          }
91          // 返回方法执行的结果
92          return obj;
93      }
94  }
```

## 3.7 基于XML

```
1   <?xml version="1.0" encoding="UTF-8"?>
```

```xml
<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
       https://www.springframework.org/schema/aop/spring-aop.xsd">

    <!--1. 将目标类交给spring管理-->
    <bean id="userService" class="com.imcode.spring.service.UserService"/>
    <!--2. 将通知/增强交给spirng管理-->
    <bean id="logAspect" class="com.imcode.spring.aspect.LogAspect"/>
    <!--3. 配置切面-->
    <aop:config>
        <!--引入增强-->
        <aop:aspect ref="logAspect">
            <aop:pointcut id="logPointcut"
               expression="execution(*
com.imcode.spring.service.UserService.login(..))"/>

            <!-- 前置通知 在方法执行之前执行-->
            <aop:before method="before" pointcut-ref="logPointcut"/>

            <!-- 返回通知 在方法返回结果之后执行 如果发生异常不再执行-->
            <aop:after-returning method="afterReturning"
                                 pointcut-ref="logPointcut" returning="result"/>

            <!--异常通知 只发生异常的时候执行-->
            <aop:after-throwing method="afterThrowing"
                                pointcut-ref="logPointcut" throwing="e"/>

            <!--后置通知 在方法执行之后执行 无论是否发生异常都执行-->
            <aop:after method="after" pointcut-ref="logPointcut"/>

            <!--环绕通知-->
            <aop:around method="around" pointcut-ref="logPointcut"/>
        </aop:aspect>
    </aop:config>
</beans>
```

# 3.8 基于注解

## spring-config.xml的配置

```xml
<!-- 启用注解扫描 -->
<context:component-scan base-package="com.imcode.spring"/>

<aop:aspectj-autoproxy proxy-target-class="false"/>
```

## 增强/通知的配置

```java
package com.imcode.spring.aspect;

import org.apache.logging.log4j.LogManager;
```

```java
import org.apache.logging.log4j.Logger;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.*;
import org.springframework.stereotype.Component;
/**
 * 增强/通知 (advice)
 */
@Component
@Aspect
public class LoginLogAspect {

    private static Logger logger = LogManager.getLogger(LoginLogAspect.class);

    @Pointcut("execution(* com.imcode.spring.service.UserService.login(..))")
    public void pointCut(){

    }

    /**
     * 前置通知
     * - 在方法执行之前执行
     * - 如果通知抛出异常，阻止方法运行
     * @param joinPoint
     */
    @Before("pointCut()")
    public void before(JoinPoint joinPoint){
        logger.info("@Before 前置通知:{}",joinPoint.getSignature().getName() );
    }

   /**
     * 返回通知
     * - 在方法返回结果之后执行
     * - 如果方法中抛出异常，该通知不执行
     * @param joinPoint  连接点 存储拦截到的方法的相关信息
     * @param result  目标方法执行后的返回值
     */
    @AfterReturning(pointcut = "pointCut()",returning = "result")
    public void afterReturning(JoinPoint joinPoint, Object result){
        // 目标方法的名称
        String methodName = joinPoint.getSignature().getName();
        // 目标方法的参数
        Object[] params = joinPoint.getArgs();
        // 目标类的对象
        Object target = joinPoint.getTarget();

//        System.out.println("目标方法的名称:" + methodName);
//        System.out.println("目标方法的参数:" + Arrays.toString(params));
//        System.out.println("目标类的对象:" + target);
//        System.out.println("方法返回值:" + result);

        logger.info("@AfterReturning 返回通知: 用户 {} 登录系统成功...",params[0]);
    }

     /**
     * 后置通知
     * - 在方法执行之后执行
     * - 无论方法中是否出现异常该通知都执行
```

```
62          * @param joinPoint
63          */
64         @After("pointCut()")
65         public void after(JoinPoint joinPoint) {
66             logger.info("@After 后置通知:{}",joinPoint.getSignature().getName() );
67         }
68 
69         /**
70          * 异常通知
71          * - 方法抛出异常后执行，
72          * - 如果方法没有抛出异常，该通知不执行
73          * @param joinPoint
74          * @param e
75          */
76         @AfterThrowing(pointcut = "pointCut()",throwing = "e")
77         public void afterThrowing(JoinPoint joinPoint, Throwable e) {
78             // 目标方法的参数
79             Object[] params = joinPoint.getArgs();
80             logger.error("@afterThrowing 异常通知: 用户 {} 登录系统异常:{}", params[0],
81                     e.getMessage());
82         }
83 
84         /**
85          * 环绕通知
86          * - 方法执行前后分别执行，可以阻止方法的执行
87          * - 必须手动执行目标方法
88          * @param joinPoint
89          */
90         public Object around(ProceedingJoinPoint joinPoint){
91             logger.info("环绕通知（@Before 前置通知）:");
92 
93             Object obj = null;
94             try {
95                 // 手动执行目标方法
96                 // obj = joinPoint.proceed();
97                 Object[] params = joinPoint.getArgs();
98                 params[1] = "111111";
99 
100                // 在运行期动态的修改传入方法参数的值
101                obj = joinPoint.proceed(params);
102 
103                logger.info("环绕通知（@AfterRunning 返回通知）:");
104 
105            } catch (Throwable e) {
106                logger.info("环绕通知（@AfterThrowing 异常通知）:");
107                e.printStackTrace();
108            } finally {
109                logger.info("环绕通知（@After 后置通知）:");
110            }
111            // 返回方法执行的结果
112            return obj;
113        }
114    }
115 
```

# 4. 切点表达式

在Spring AOP中，需要使用AspectJ的切点表达式语言来定义切点。下表列出了Spring AOP所支持的AspectJ切点指示器：

## 4.1 execution()

`execution()` 用于描述哪些方法定义成切点方法。

基本语法：

```
execution(修饰符  返回值  包.类.方法名(参数) throws 异常)
```

## 4.2 方法修饰符

方法修饰符，一般省略

- `public` 公共方法
- `*` 任意方法

```
public
```

## 4.3 方法返回值

方法返回值，不能省略

- `void` 没有返回值的方法
- `String` 返回值是字符串
- `*` 任意方法

```
* 方法名称() //任意返回值的方法
```

## 4.4 包

- `com.imcode.app`
  - 固定包
- `com.imcode.app.*.service`
  - 包下面子包任意
  - 例如： com.imcode.app. `common` .service
  - 例如： com.imcode.app. `system` .service
- `com.imcode.app..service`
  - app 包下面的所有子包（含自己）
  - 例如: com.imcode. `app` .service
  - 例如: com.imcode.app. `common` .service
  - 例如: com.imcode.app. `common.user` .service

## 4.5 类

- `UserService` 指定类
- 例如: com.imcode.app.service.UserService
- `*Service`

```
com.imcode.app.service.*Service
```

  - 以 `Service` 结尾
  - 例如：com.imcode.app.service. `UserService`
  - 例如：com.imcode.app.service. `RoleService`
- `User*`

```
com.imcode.app.service.User*
```

  - 以 `User` 开头
  - 例如：com.imcode.app.service. `UserService`
  - 例如：com.imcode.app.service. `UserRoleService`
- `*` 任意

## 4.6 方法名称

方法名，不能省略

- `getById` 固定方法
- `get*` 以 `get` 开头
- `*do` 以 `do` 结尾
- `*` 任意方法

```
execution( * com.imcode.spring.service.*Service.*())
```

## 4.7 方法参数

- `()` 无参
- `(int)` 一个整型参数
- `(int ,String)` 一个整形参数，一个字符串参数
- `(..)` 参数任意
- `(*)` 一个参数任意类型
- `(*, *)` 两个参数参数任意类型

```
execution( * com.imcode.spring.service.*Service.*(..))
```

throws ,可省略，一般不写